CHAPMAN & HALL/CRC
CRYPTOGRAPHY AND NETWORK SECURITY

INTRODUCTION TO MODERN CRYPTOGRAPHY

Jonathan Katz
Yehuda Lindell

Chapman & Hall/CRC
Taylor & Francis Group

CHAPMAN & HALL/CRC
CRYPTOGRAPHY AND NETWORK SECURITY

Introduction to
Modern Cryptography

CHAPMAN & HALL/CRC
CRYPTOGRAPHY AND NETWORK SECURITY

Introduction to
Modern Cryptography

Jonathan Katz
Yehuda Lindell

Chapman & Hall/CRC
Taylor & Francis Group
Boca Raton London New York

Chapman & Hall/CRC is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2008 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20110713

International Standard Book Number-13: 978-1-4200-1075-6 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

CHAPMAN & HALL/CRC CRYPTOGRAPHY AND NETWORK SECURITY

Series Editor
Douglas R. Stinson

Published Titles

Jonathan Katz and Yehuda Lindell, Introduction to Modern Cryptography

Forthcoming Titles

Burton Rosenberg, Handbook of Financial Cryptography

Maria Isabel Vasco, Spyros Magliveras, and Rainer Steinwandt, Group Theoretic Cryptography

Shiu-Kai Chin and Susan Beth Older, A Mathematical Introduction to Access Control

Preface

This book presents the basic paradigms and principles of modern cryptography. It is designed to serve as a textbook for undergraduate- or graduate-level courses in cryptography (in computer science or mathematics departments), as a general introduction suitable for self-study (especially for beginning graduate students), and as a reference for students, researchers, and practitioners.

There are numerous other cryptography textbooks available today, and the reader may rightly ask whether another book on the subject is needed. We would not have written this book if the answer to that question were anything other than an unequivocal *yes*. The novelty of this book — and what, in our opinion, distinguishes it from all other books currently available — is that it provides a *rigorous* treatment of modern cryptography in an *accessible* manner appropriate for an introduction to the topic.

As mentioned, our focus is on *modern* (post-1980s) cryptography, which is distinguished from classical cryptography by its emphasis on definitions, precise assumptions, and rigorous proofs of security. We briefly discuss each of these in turn (these principles are explored in greater detail in Chapter 1):

- **The central role of definitions:** A key intellectual contribution of modern cryptography has been the recognition that *formal definitions of security are an essential first step in the design of any cryptographic primitive or protocol*. The reason, in retrospect, is simple: if you don't know what it is you are trying to achieve, how can you hope to know when you have achieved it? As we will see in this book, cryptographic definitions of security are quite strong and — at first glance — may appear impossible to achieve. One of the most amazing aspects of cryptography is that (under mild and widely-believed assumptions) efficient constructions satisfying such strong definitions can be proven to exist.

- **The importance of formal and precise assumptions:** As will be explained in Chapters 2 and 3, many cryptographic constructions cannot currently be proven secure in an unconditional sense. Security often relies, instead, on some widely-believed (albeit unproven) assumption. The modern cryptographic approach dictates that *any such assumption must be clearly stated and unambiguously defined*. This not only allows for objective evaluation of the assumption but, more importantly, enables rigorous proofs of security as described next.

- **The possibility of rigorous proofs of security:** The previous two ideas lead naturally to the current one, which is the realization that *cryp-*

tographic constructions can be proven secure with respect to a clearly-stated definition of security and relative to a well-defined cryptographic assumption. This is the essence of modern cryptography, and what has transformed cryptography from an art to a science.

The importance of this idea cannot be over-emphasized. Historically, cryptographic schemes were designed in a largely ad-hoc fashion, and were deemed to be secure if the designers themselves could not find any attacks. In contrast, modern cryptography promotes the design of schemes with formal, mathematical proofs of security in well-defined models. Such schemes are *guaranteed* to be secure unless the underlying assumption is false (or the security definition did not appropriately model the real-world security concerns). By relying on long-standing assumptions (e.g., the assumption that "factoring is hard"), it is thus possible to obtain schemes that are extremely unlikely to be broken.

A unified approach. The above contributions of modern cryptography are relevant not only to the "theory of cryptography" community. The importance of precise definitions is, by now, widely understood and appreciated by those in the security community who use cryptographic tools to build secure systems, and rigorous proofs of security have become one of the requirements for cryptographic schemes to be standardized. As such, we do not separate "applied cryptography" from "provable security"; rather, we present practical and widely-used constructions along with precise statements (and, most of the time, a proof) of what definition of security is achieved.

Guide to Using this Book

This section is intended primarily for instructors seeking to adopt this book for their course, though the student picking up this book on his or her own may also find it a useful overview of the topics that will be covered.

Required background. This book uses definitions, proofs, and mathematical concepts, and therefore requires some mathematical maturity. In particular, the reader is assumed to have had some exposure to proofs at the college level, say in an upper-level mathematics course or a course on discrete mathematics, algorithms, or computability theory. Having said this, we have made a significant effort to simplify the presentation and make it generally accessible. It is our belief that this book is not more difficult than analogous textbooks that are less rigorous. On the contrary, we believe that (to take one example) once security goals are clearly formulated, it often becomes easier to understand the design choices made in a particular construction.

We have structured the book so that the only formal prerequisites are a course in algorithms and a course in discrete mathematics. Even here we rely on very little material: specifically, we assume some familiarity with basic probability and big-$\mathcal{O}$ notation, modular arithmetic, and the idea of equating

efficient algorithms with those running in polynomial time. These concepts are reviewed in Appendix A and/or when first used in the book.

Suggestions for course organization. The core material of this book, which we strongly recommend should be covered in any introductory course on cryptography, consists of the following (starred sections are excluded in what follows; see further discussion regarding starred material below):

- Chapters 1–4 (through Section 4.6), discussing classical cryptography, modern cryptography, and the basics of private-key cryptography (both private-key encryption and message authentication).
- Chapter 5, illustrating basic design principles for block ciphers and including material on the widely-used block ciphers DES and AES.[1]
- Chapter 7, introducing concrete mathematical problems believed to be "hard", and providing the number-theoretic background needed to understand the RSA, Diffie-Hellman, and El Gamal cryptosystems. This chapter also gives the first examples of how number-theoretic assumptions are used in cryptography.
- Chapters 9 and 10, motivating the public-key setting and discussing public-key encryption (including RSA-based schemes and El Gamal encryption).
- Chapter 12, describing digital signature schemes.
- Sections 13.1 and 13.3, introducing the random oracle model and the RSA-FDH signature scheme.

We believe that this core material — possibly omitting some of the more in-depth discussion and proofs — can be covered in a 30–35-hour undergraduate course. Instructors with more time available could proceed at a more leisurely pace, e.g., giving details of all proofs and going more slowly when introducing the underlying group theory and number-theoretic background. Alternatively, additional topics could be incorporated as discussed next.

Those wishing to cover additional material, in either a longer course or a faster-paced graduate course, will find that the book has been structured to allow flexible incorporation of other topics as time permits (and depending on the instructor's interests). Specifically, some of the chapters and sections are starred (*). These sections are not less important in any way, but arguably do not constitute "core material" for an introductory course in cryptography. As made evident by the course outline just given (which does not include any starred material), starred chapters and sections may be skipped — or covered at any point subsequent to their appearance in the book — without affecting

[1] Although we consider this to be core material, it is not used in the remainder of the book and so this chapter can be skipped if desired.

the flow of the course. In particular, we have taken care to ensure that none of the later un-starred material depends on any starred material. For the most part, the starred chapters also do not depend on each other (and when they do, this dependence is explicitly noted).

We suggest the following from among the starred topics for those wishing to give their course a particular flavor:

- *Theory:* A more theoretically-inclined course could include material from Section 3.2.2 (building to a definition of semantic security for encryption); Sections 4.8 and 4.9 (dealing with stronger notions of security for private-key encryption); Chapter 6 (introducing one-way functions and hard-core bits, and constructing pseudorandom generators and pseudorandom functions/permutations starting from any one-way permutation); Section 10.7 (constructing public-key encryption from trapdoor permutations); Chapter 11 (describing the Goldwasser-Micali, Rabin, and Paillier encryption schemes); and Section 12.6 (showing a signature scheme that does not rely on random oracles).

- *Applications:* An instructor wanting to emphasize practical aspects of cryptography is highly encouraged to cover Section 4.7 (describing HMAC) and all of Chapter 13 (giving cryptographic constructions in the random oracle model).

- *Mathematics:* A course directed at students with a strong mathematics background — or taught by someone who enjoys this aspect of cryptography — could incorporate some of the more advanced number theory from Chapter 7 (e.g., the Chinese remainder theorem and/or elliptic-curve groups); all of Chapter 8 (algorithms for factoring and computing discrete logarithms); and selections from Chapter 11 (describing the Goldwasser-Micali, Rabin, and Paillier encryption schemes along with the necessary number-theoretic background).

Comments and Errata

Our goal in writing this book was to make modern cryptography accessible to a wide audience outside the "theoretical computer science" community. We hope you will let us know whether we have succeeded. In particular, we are always more than happy to receive feedback on this book, especially constructive comments telling us how the book can be improved. We hope there are no errors or typos in the book; if you do find any, however, we would greatly appreciate it if you let us know. (A list of known errata will be maintained at `http://www.cs.umd.edu/~jkatz/imc.html`.) You can email your comments and errata to `jkatz@cs.umd.edu` and `lindell@cs.biu.ac.il`; please put "Introduction to Modern Cryptography" in the subject line.

Acknowledgements

Jonathan Katz: I am indebted to Zvi Galil, Moti Yung, and Rafail Ostrovsky for their help, guidance, and support throughout my career. This book would never have come to be without their contributions to my development. I would also like to thank my colleagues with whom I have enjoyed numerous discussions on the "right" approach to writing a cryptography textbook. My work on this project was supported in part by the National Science Foundation under Grants #0627306, #0447075, and #0310751. Any opinions, findings, and conclusions or recommendations expressed in this book are my own, and do not necessarily reflect the views of the National Science Foundation.

Yehuda Lindell: I wish to first and foremost thank Oded Goldreich and Moni Naor for introducing me to the world of cryptography. Their influence is felt until today and will undoubtedly continue to be felt in the future. There are many, many other people who have also had considerable influence over the years and instead of mentioning them all, I will just say *thank you* — you know who you are.

We both thank Zoe Bermant for producing the figures used in this book; David Wagner for answering questions related to block ciphers and their cryptanalysis; and Salil Vadhan and Alon Rosen for experimenting with this text in an introductory course on cryptography at Harvard University and providing us with valuable feedback. We would also like to extend our gratitude to those who read and commented on earlier drafts of this book and to those who sent us corrections to previous printings: Adam Bender, Chiu-Yuen Koo, Yair Dombb, Michael Fuhr, William Glenn, S. Dov Gordon, Carmit Hazay, Eyal Kushilevitz, Avivit Levy, Matthew Mah, Ryan Murphy, Steve Myers, Martin Paraskevov, Eli Quiroz, Jason Rogers, Rui Xue, Dicky Yan, Arkady Yerukhimovich, and Hila Zarosim. Their comments have greatly improved the book and helped minimize the number of errors. We are extremely grateful to all those who encouraged us to write this book, and concurred with our feeling that a book of this nature is badly needed.

Finally, we thank our (respective) wives and children for all their support and understanding during the many hours, days, and months that we have spent on this project.

To our wives and children

Contents

I Introduction and Classical Cryptography 1

1 Introduction 3
1.1 Cryptography and Modern Cryptography 3
1.2 The Setting of Private-Key Encryption 4
1.3 Historical Ciphers and Their Cryptanalysis 9
1.4 The Basic Principles of Modern Cryptography 18
1.4.1 Principle 1 – Formulation of Exact Definitions 18
1.4.2 Principle 2 – Reliance on Precise Assumptions 24
1.4.3 Principle 3 – Rigorous Proofs of Security 26
References and Additional Reading 27
Exercises 27

2 Perfectly-Secret Encryption 29
2.1 Definitions and Basic Properties 29
2.2 The One-Time Pad (Vernam's Cipher) 34
2.3 Limitations of Perfect Secrecy 36
2.4 * Shannon's Theorem 37
2.5 Summary 40
References and Additional Reading 40
Exercises 41

II Private-Key (Symmetric) Cryptography 45

3 Private-Key Encryption and Pseudorandomness 47
3.1 A Computational Approach to Cryptography 47
3.1.1 The Basic Idea of Computational Security 48
3.1.2 Efficient Algorithms and Negligible Success Probability 54
3.1.3 Proofs by Reduction 58
3.2 Defining Computationally-Secure Encryption 60
3.2.1 The Basic Definition of Security 61
3.2.2 * Properties of the Definition 64
3.3 Pseudorandomness 69
3.4 Constructing Secure Encryption Schemes 72
3.4.1 A Secure Fixed-Length Encryption Scheme 72
3.4.2 Handling Variable-Length Messages 76
3.4.3 Stream Ciphers and Multiple Encryptions 77

3.5 Security Against Chosen-Plaintext Attacks (CPA) 82
3.6 Constructing CPA-Secure Encryption Schemes 85
3.6.1 Pseudorandom Functions 86
3.6.2 CPA-Secure Encryption from Pseudorandom Functions 89
3.6.3 Pseudorandom Permutations and Block Ciphers . . . 94
3.6.4 Modes of Operation 96
3.7 Security Against Chosen-Ciphertext Attacks (CCA) 103
References and Additional Reading 105
Exercises 106

4 Message Authentication Codes and Collision-Resistant Hash Functions 111
4.1 Secure Communication and Message Integrity 111
4.2 Encryption vs. Message Authentication 112
4.3 Message Authentication Codes – Definitions 114
4.4 Constructing Secure Message Authentication Codes 118
4.5 CBC-MAC 125
4.6 Collision-Resistant Hash Functions 127
4.6.1 Defining Collision Resistance 128
4.6.2 Weaker Notions of Security for Hash Functions 130
4.6.3 A Generic "Birthday" Attack 131
4.6.4 The Merkle-Damgård Transform 133
4.6.5 Collision-Resistant Hash Functions in Practice 136
4.7 * NMAC and HMAC 138
4.7.1 Nested MAC (NMAC) 138
4.7.2 HMAC 141
4.8 * Constructing CCA-Secure Encryption Schemes 144
4.9 * Obtaining Privacy and Message Authentication 148
References and Additional Reading 154
Exercises 155

5 Practical Constructions of Pseudorandom Permutations (Block Ciphers) 159
5.1 Substitution-Permutation Networks 162
5.2 Feistel Networks 170
5.3 DES – The Data Encryption Standard 173
5.3.1 The Design of DES 173
5.3.2 Attacks on Reduced-Round Variants of DES 176
5.3.3 The Security of DES 179
5.4 Increasing the Key Length of a Block Cipher 181
5.5 AES – The Advanced Encryption Standard 185
5.6 Differential and Linear Cryptanalysis – A Brief Look 187
Additional Reading and References 189
Exercises 189

6 * Theoretical Constructions of Pseudorandom Objects 193
6.1 One-Way Functions 194
6.1.1 Definitions 194
6.1.2 Candidate One-Way Functions 197
6.1.3 Hard-Core Predicates 198
6.2 Overview: From One-Way Functions to Pseudorandomness . 200
6.3 A Hard-Core Predicate for Any One-Way Function 202
6.3.1 A Simple Case 202
6.3.2 A More Involved Case 203
6.3.3 The Full Proof 208
6.4 Constructing Pseudorandom Generators 213
6.4.1 Pseudorandom Generators with Minimal Expansion . 214
6.4.2 Increasing the Expansion Factor 215
6.5 Constructing Pseudorandom Functions 221
6.6 Constructing (Strong) Pseudorandom Permutations 225
6.7 Necessary Assumptions for Private-Key Cryptography 227
6.8 A Digression – Computational Indistinguishability 232
6.8.1 Pseudorandomness and Pseudorandom Generators . . 233
6.8.2 Multiple Samples 234
References and Additional Reading 237
Exercises 237

III Public-Key (Asymmetric) Cryptography 241

7 Number Theory and Cryptographic Hardness Assumptions 243
7.1 Preliminaries and Basic Group Theory 245
7.1.1 Primes and Divisibility 246
7.1.2 Modular Arithmetic 248
7.1.3 Groups 250
7.1.4 The Group $\mathbb{Z}_N^*$ 254
7.1.5 * Isomorphisms and the Chinese Remainder Theorem 256
7.2 Primes, Factoring, and RSA 261
7.2.1 Generating Random Primes 262
7.2.2 * Primality Testing 265
7.2.3 The Factoring Assumption 271
7.2.4 The RSA Assumption 271
7.3 Assumptions in Cyclic Groups 274
7.3.1 Cyclic Groups and Generators 274
7.3.2 The Discrete Logarithm and Diffie-Hellman Assumptions 277
7.3.3 Working in (Subgroups of) $\mathbb{Z}_p^*$ 281
7.3.4 * Elliptic Curve Groups 282
7.4 Cryptographic Applications of Number-Theoretic Assumptions 287
7.4.1 One-Way Functions and Permutations 287
7.4.2 Constructing Collision-Resistant Hash Functions . . . 290

References and Additional Reading 293
Exercises 294

8 * Factoring and Computing Discrete Logarithms 297
8.1 Algorithms for Factoring 297
8.1.1 Pollard's $p-1$ Method 298
8.1.2 Pollard's Rho Method 301
8.1.3 The Quadratic Sieve Algorithm 303
8.2 Algorithms for Computing Discrete Logarithms 305
8.2.1 The Baby-Step/Giant-Step Algorithm 307
8.2.2 The Pohlig-Hellman Algorithm 309
8.2.3 The Discrete Logarithm Problem in $\mathbb{Z}_N$ 310
8.2.4 The Index Calculus Method 311
References and Additional Reading 313
Exercises 314

9 Private-Key Management and the Public-Key Revolution 315
9.1 Limitations of Private-Key Cryptography 315
9.2 A Partial Solution – Key Distribution Centers 317
9.3 The Public-Key Revolution 320
9.4 Diffie-Hellman Key Exchange 324
References and Additional Reading 330
Exercises 331

10 Public-Key Encryption 333
10.1 Public-Key Encryption – An Overview 333
10.2 Definitions 336
10.2.1 Security against Chosen-Plaintext Attacks 337
10.2.2 Multiple Encryptions 340
10.3 Hybrid Encryption 347
10.4 RSA Encryption 355
10.4.1 "Textbook RSA" and its Insecurity 355
10.4.2 Attacks on Textbook RSA 359
10.4.3 Padded RSA 362
10.5 The El Gamal Encryption Scheme 364
10.6 Security Against Chosen-Ciphertext Attacks 369
10.7 * Trapdoor Permutations 373
10.7.1 Definition 374
10.7.2 Public-Key Encryption from Trapdoor Permutations . 375
References and Additional Reading 378
Exercises 379

11 * Additional Public-Key Encryption Schemes **385**
11.1 The Goldwasser-Micali Encryption Scheme 386
11.1.1 Quadratic Residues Modulo a Prime 386
11.1.2 Quadratic Residues Modulo a Composite 389
11.1.3 The Quadratic Residuosity Assumption 392
11.1.4 The Goldwasser-Micali Encryption Scheme 394
11.2 The Rabin Encryption Scheme 397
11.2.1 Computing Modular Square Roots 397
11.2.2 A Trapdoor Permutation Based on Factoring 402
11.2.3 The Rabin Encryption Scheme 406
11.3 The Paillier Encryption Scheme 408
11.3.1 The Structure of $\mathbb{Z}_{N^2}^*$ 409
11.3.2 The Paillier Encryption Scheme 411
11.3.3 Homomorphic Encryption 416
References and Additional Reading 418
Exercises 418

12 Digital Signature Schemes **421**
12.1 Digital Signatures – An Overview 421
12.2 Definitions 423
12.3 RSA Signatures 426
12.3.1 "Textbook RSA" and its Insecurity 426
12.3.2 Hashed RSA 428
12.4 The "Hash-and-Sign" Paradigm 429
12.5 Lamport's One-Time Signature Scheme 432
12.6 * Signatures from Collision-Resistant Hashing 435
12.6.1 "Chain-Based" Signatures 436
12.6.2 "Tree-Based" Signatures 439
12.7 The Digital Signature Standard (DSS) 445
12.8 Certificates and Public-Key Infrastructures 446
References and Additional Reading 453
Exercises 454

13 Public-Key Cryptosystems in the Random Oracle Model **457**
13.1 The Random Oracle Methodology 458
13.1.1 The Random Oracle Model in Detail 459
13.1.2 Is the Random Oracle Methodology Sound? 465
13.2 Public-Key Encryption in the Random Oracle Model 469
13.2.1 Security Against Chosen-Plaintext Attacks 469
13.2.2 Security Against Chosen-Ciphertext Attacks 473
13.2.3 OAEP 479
13.3 Signatures in the Random Oracle Model 481
References and Additional Reading 486
Exercises 486

Index of Common Notation **489**

A Mathematical Background **493**
A.1 Identities and Inequalities 493
A.2 Asymptotic Notation 493
A.3 Basic Probability 494
A.4 The "Birthday" Problem 496

B Supplementary Algorithmic Number Theory **499**
B.1 Integer Arithmetic 501
B.1.1 Basic Operations 501
B.1.2 The Euclidean and Extended Euclidean Algorithms . 502
B.2 Modular Arithmetic 504
B.2.1 Basic Operations 504
B.2.2 Computing Modular Inverses 505
B.2.3 Modular Exponentiation 505
B.2.4 Choosing a Random Group Element 508
B.3 * Finding a Generator of a Cyclic Group 512
B.3.1 Group-Theoretic Background 512
B.3.2 Efficient Algorithms 513
References and Additional Reading 515
Exercises 515

References **517**

Index **529**

Part I

Introduction and Classical Cryptography

Chapter 1

Introduction

1.1 Cryptography and Modern Cryptography

The Concise Oxford Dictionary (2006) defines cryptography as *the art of writing or solving codes*. This definition may be historically accurate, but it does not capture the essence of modern cryptography. First, it focuses solely on the problem of secret communication. This is evidenced by the fact that the definition specifies "codes", elsewhere defined as "a system of pre-arranged signals, especially used to ensure secrecy in transmitting messages". Second, the definition refers to cryptography as an art form. Indeed, until the 20th century (and arguably until late in that century), cryptography was an art. Constructing good codes, or breaking existing ones, relied on creativity and personal skill. There was very little theory that could be relied upon and there was not even a well-defined notion of what constitutes a good code.

In the late 20th century, this picture of cryptography radically changed. A rich theory emerged, enabling the rigorous study of cryptography as a *science*. Furthermore, the field of cryptography now encompasses much more than secret communication. For example, it deals with the problems of message authentication, digital signatures, protocols for exchanging secret keys, authentication protocols, electronic auctions and elections, digital cash and more. In fact, modern cryptography can be said to be concerned with problems that may arise in *any* distributed computation that may come under internal or external attack. Without attempting to provide a perfect definition of modern cryptography, we would say that it is *the scientific study of techniques for securing digital information, transactions, and distributed computations.*

Another very important difference between classical cryptography (say, before the 1980s) and modern cryptography relates to who uses it. Historically, the major consumers of cryptography were military and intelligence organizations. Today, however, cryptography is everywhere! Security mechanisms that rely on cryptography are an integral part of almost any computer system. Users (often unknowingly) rely on cryptography every time they access a secured website. Cryptographic methods are used to enforce access control in multi-user operating systems, and to prevent thieves from extracting trade secrets from stolen laptops. Software protection methods employ encryption, authentication, and other tools to prevent copying. The list goes on and on.

In short, cryptography has gone from an art form that dealt with secret communication for the military to a science that helps to secure systems for ordinary people all across the globe. This also means that cryptography is becoming a more and more central topic within computer science.

The focus of this book is *modern* cryptography. Yet we will begin our study by examining the state of cryptography before the changes mentioned above. Besides allowing us to ease into the material, it will also provide an understanding of where cryptography has come from so that we can later appreciate how much it has changed. The study of "classical cryptography" — replete with ad-hoc constructions of codes, and relatively simple ways to break them — serves as good motivation for the more rigorous approach that we will be taking in the rest of the book.[1]

1.2 The Setting of Private-Key Encryption

As noted above, cryptography was historically concerned with secret communication. Specifically, cryptography was concerned with the construction of *ciphers* (now called *encryption schemes*) for providing secret communication between two parties sharing some information in advance. The setting in which the communicating parties share some secret information in advance is now known as the *private-key* (or the *symmetric-key*) setting. Before describing some historical ciphers, we discuss the private-key setting and encryption in more general terms.

In the private-key setting, two parties share some secret information called a *key*, and use this key when they wish to communicate secretly with each other. A party sending a message uses the key to *encrypt* (or "scramble") the message before it is sent, and the receiver uses the same key to *decrypt* (or "unscramble") and recover the message upon receipt. The message itself is called the *plaintext*, and the "scrambled" information that is actually transmitted from the sender to the receiver is called the *ciphertext*; see Figure 1.1. The shared key serves to distinguish the communicating parties from any other parties who may be eavesdropping on their communication (assumed to take place over a public channel).

In this setting, the same key is used to convert the plaintext into a ciphertext and back. This explains why this setting is also known as the *symmetric-key* setting, where the symmetry lies in the fact that both parties hold the same key which is used for both encryption and decryption. This is in contrast to

[1] This is our primary intent in presenting this material and, as such, this chapter should not be taken as a representative historical account. The reader interested in the history of cryptography should consult the references at the end of this chapter.

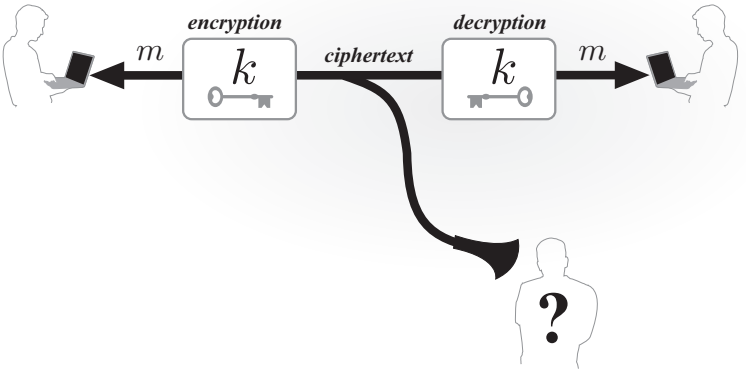

FIGURE 1.1: The basic setting of private-key encryption.

the setting of *asymmetric* encryption (introduced in Chapter 9), where the sender and receiver do not share any secrets and different keys are used for encryption and decryption. The private-key setting is the classic one, as we will see later in this chapter.

An implicit assumption in any system using private-key encryption is that the communicating parties have some way of initially sharing a key in a secret manner. (Note that if one party simply sends the key to the other over the public channel, an eavesdropper obtains the key too!) In military settings, this is not a severe problem because communicating parties are able to physically meet in a secure location in order to agree upon a key. In many modern settings, however, parties cannot arrange any such physical meeting. As we will see in Chapter 9, this is a source of great concern and actually limits the applicability of cryptographic systems that rely solely on private-key methods. Despite this, there are still many settings where private-key methods suffice and are in wide use; one example is disk encryption, where the *same* user (at different points in time) uses a fixed secret key to both write to and read from the disk. As we will explore further in Chapter 10, private-key encryption is also widely used in conjunction with asymmetric methods.

The syntax of encryption. A private-key encryption scheme is comprised of three algorithms: the first is a procedure for generating keys, the second a procedure for encrypting, and the third a procedure for decrypting. These have the following functionality:

1. The *key-generation algorithm* Gen is a probabilistic algorithm that outputs a key k chosen according to some distribution that is determined by the scheme.

2. The *encryption algorithm* Enc takes as input a key k and a plaintext message m and outputs a ciphertext c. We denote by $\mathsf{Enc}_k(m)$ the encryption of the plaintext m using the key k.

3. The *decryption algorithm* Dec takes as input a key k and a ciphertext c and outputs a plaintext m. We denote the decryption of the ciphertext c using the key k by $\mathsf{Dec}_k(c)$.

The set of all possible keys output by the key-generation algorithm is called the *key space* and is denoted by $\mathcal{K}$. Almost always, Gen simply chooses a key uniformly at random from the key space (in fact, one can assume without loss of generality that this is the case). The set of all "legal" messages (i.e., those supported by the encryption algorithm) is denoted $\mathcal{M}$ and is called the *plaintext* (or *message*) *space*. Since any ciphertext is obtained by encrypting some plaintext under some key, the sets $\mathcal{K}$ and $\mathcal{M}$ together define a set of all possible ciphertexts denoted by $\mathcal{C}$. An encryption scheme is fully defined by specifying the three algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and the plaintext space $\mathcal{M}$.

The basic correctness requirement of any encryption scheme is that for every key k output by Gen and every plaintext message $m \in \mathcal{M}$, it holds that

$$\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m.$$

In words, decrypting a ciphertext (using the appropriate key) yields the original message that was encrypted.

Recapping our earlier discussion, an encryption scheme would be used by two parties who wish to communicate as follows. First, Gen is run to obtain a key k that the parties share. When one party wants to send a plaintext m to the other, he computes $c := \mathsf{Enc}_k(m)$ and sends the resulting ciphertext c over the public channel to the other party.[2] Upon receiving c, the other party computes $m := \mathsf{Dec}_k(c)$ to recover the original plaintext.

Keys and Kerckhoffs' principle. As is clear from the above formulation, if an eavesdropping adversary knows the algorithm Dec as well as the key k shared by the two communicating parties, then that adversary will be able to decrypt all communication between these parties. It is for this reason that the communicating parties must share the key k secretly, and keep k completely secret from everyone else. But maybe they should keep the decryption algorithm Dec a secret, too? For that matter, perhaps all the algorithms constituting the encryption scheme (i.e., Gen and Enc as well) should be kept secret? (Note that the plaintext space $\mathcal{M}$ is typically assumed to be known, e.g., it may consist of English-language sentences.)

In the late 19th century, Auguste Kerckhoffs gave his opinion on this matter in a paper he published outlining important design principles for military

[2] Throughout the book, we use ":=" to denote the assignment operation. A list of common notation can be found in the back of the book.

ciphers. One of the most important of these principles (now known simply as Kerckhoffs' principle) is the following:

> *The cipher method must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience.*

In other words, the encryption scheme itself should *not* be kept secret, and so only the key should constitute the secret information shared by the communicating parties.

Kerckhoffs' intention was that an encryption scheme should be designed so as to be secure *even if* an adversary knows the details of all the component algorithms of the scheme, as long as the adversary doesn't know the key being used. Stated differently, Kerckhoffs' principle demands that *security rely solely on the secrecy of the key.* But why?

There are three primary arguments in favor of Kerckhoffs' principle. The first is that it is much easier for the parties to maintain secrecy of a short key than to maintain secrecy of an algorithm. It is easier to share a short (say, 100-bit) string and store this string securely than it is to share and securely store a program that is thousands of times larger. Furthermore, details of an algorithm can be leaked (perhaps by an insider) or learned through reverse engineering; this is unlikely when the secret information takes the form of a randomly-generated string.

A second argument in favor of Kerckhoffs' principle is that in case the key *is* exposed, it will be much easier for the honest parties to change the key than to replace the algorithm being used. Actually, it is good security practice to refresh a key frequently even when it has not been exposed, and it would be much more cumbersome to replace the software being used instead.

Finally, in case many pairs of people (say, within a company) need to encrypt their communication, it will be significantly easier for all parties to use the same algorithm/program, but different keys, than for everyone to use a different program (which would furthermore depend on the party with whom they are communicating).

Today, Kerckhoffs' principle is understood as not only advocating that security should not rely on secrecy of the algorithms being used, but also demanding that these algorithms be made *public*. This stands in stark contrast to the notion of "security by obscurity" which is the idea that improved security can be achieved by keeping a cryptographic algorithm hidden. Some of the advantages of "open cryptographic design", where algorithm specifications are made public, include the following:

1. Published designs undergo public scrutiny and are therefore likely to be stronger. Many years of experience have demonstrated that it is very difficult to construct good cryptographic schemes. Therefore, our confidence in the security of a scheme is much higher if it has been extensively studied (by experts other than the designers of the scheme themselves) and no weaknesses have been found.

2. It is better for security flaws, if they exist, to be revealed by "ethical hackers" (leading, hopefully, to the system being fixed) rather than having these flaws be known only to malicious parties.

3. If the security of the system relies on the secrecy of the algorithm, then reverse engineering of the code (or leakage by industrial espionage) poses a serious threat to security. This is in contrast to the secret key which is not part of the code, and so is not vulnerable to reverse engineering.

4. Public design enables the establishment of standards.

As simple and obvious as it may sound, the principle of open cryptographic design (i.e., Kerckhoffs' principle) is ignored over and over again with disastrous results. It is very dangerous to use a proprietary algorithm (i.e., a non-standardized algorithm that was designed in secret by some company), and only publicly tried and tested algorithms should be used. Fortunately, there are enough good algorithms that are standardized and not patented, so that there is no reason whatsoever today to use something else.

Attack scenarios. We wrap up our general discussion of encryption with a brief discussion of some basic types of attacks against encryption schemes. In order of severity, these are:

- Ciphertext-only attack: This is the most basic type of attack and refers to the scenario where the adversary just observes a ciphertext (or multiple ciphertexts) and attempts to determine the underlying plaintext (or plaintexts).

- Known-plaintext attack: Here, the adversary learns one or more pairs of plaintexts/ciphertexts encrypted under the same key. The aim of the adversary is then to determine the plaintext that was encrypted in some *other* ciphertext (for which it does not know the corresponding plaintext).

- Chosen-plaintext attack: In this attack, the adversary has the ability to obtain the encryption of plaintexts of its choice. It then attempts to determine the plaintext that was encrypted in some other ciphertext.

- Chosen-ciphertext attack: The final type of attack is one where the adversary is even given the capability to obtain the decryption of ciphertexts of its choice. The adversary's aim, once again, is to determine the plaintext that was encrypted in some other ciphertext (whose decryption the adversary is unable to obtain directly).

The first two types of attacks are *passive* in that the adversary just receives some ciphertexts (and possibly some corresponding plaintexts as well) and then launches its attack. In contrast, the last two types of attacks are *active* in that the adversary can adaptively ask for encryptions and/or decryptions of its choice.

The first two attacks described above are clearly realistic. A ciphertext-only attack is the easiest to carry out in practice; the only thing the adversary needs is to eavesdrop on the public communication line over which encrypted messages are sent. In a known-plaintext attack it is assumed that the adversary somehow also obtains the plaintext messages corresponding to the ciphertexts that it viewed. This is often realistic because not all encrypted messages are confidential, at least not indefinitely. As a trivial example, two parties may always encrypt a "hello" message whenever they begin communicating. As a more complex example, encryption may be used to keep quarterly earnings results secret until their release date. In this case, anyone eavesdropping and obtaining the ciphertext will later obtain the corresponding plaintext. Any reasonable encryption scheme must therefore remain secure against an adversary that can launch a known-plaintext attack.

The two latter active attacks may seem somewhat strange and require justification. (When do parties encrypt and decrypt whatever an adversary wishes?) We defer a more detailed discussion of these attacks to the place in the text where security against these attacks is formally defined: Section 3.5 for chosen-plaintext attacks and Section 3.7 for chosen-ciphertext attacks.

Different applications of encryption may require the encryption scheme to be resilient to different types of attacks. It is not always the case that an encryption scheme secure against the "strongest" type of attack should be used, since it may be less efficient than an encryption scheme secure against "weaker" attacks. Therefore, the latter may be preferred if it suffices for the application at hand.

1.3 Historical Ciphers and Their Cryptanalysis

In our study of "classical cryptography" we will examine some historical ciphers and show that they are completely insecure. As stated earlier, our main aims in presenting this material are (1) to highlight the weaknesses of an "ad-hoc" approach to cryptography, and thus motivate the modern, rigorous approach that will be discussed in the following section, and (2) to demonstrate that "simple approaches" to achieving secure encryption are unlikely to succeed, and show why this is the case. Along the way, we will present some central principles of cryptography which can be learned from the weaknesses of these historical schemes.

In this section (and this section only), plaintext characters are written in `lower case` and ciphertext characters are written in `UPPER CASE`. When describing attacks on schemes, we always apply Kerckhoffs' principle and assume that the scheme is known to the adversary (but the key being used is not).

Caesar's cipher. One of the oldest recorded ciphers, known as Caesar's cipher, is described in "De Vita Caesarum, Divus Iulius" ("The Lives of the Caesars, The Deified Julius"), written in approximately 110 C.E.:

> *There are also letters of his to Cicero, as well as to his intimates on private affairs, and in the latter, if he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.*

That is, Julius Caesar encrypted by rotating the letters of the alphabet by 3 places: `a` was replaced with `D`, `b` with `E`, and so on. Of course, at the end of the alphabet, the letters wrap around and so `x` was replaced with `A`, `y` with `B`, and `z` with `C`. For example, the short message `begin the attack now`, with spaces removed, would be encrypted as:

```
EHJLQWKHDWWDFNQRZ
```

making it unintelligible.

An immediate problem with this cipher is that the method is *fixed*. Thus, anyone learning how Caesar encrypted his messages would be able to decrypt effortlessly. This can be seen also if one tries to fit Caesar's cipher into the syntax of encryption described earlier: the key-generation algorithm Gen is trivial (that is, it does nothing) and there is no secret key to speak of.

Interestingly, a variant of this cipher called ROT-13 (where the shift is 13 places instead of 3) is widely used nowadays in various online forums. It is understood that this does not provide any cryptographic security, and ROT-13 is used merely to ensure that the text (say, a movie spoiler) is unintelligible unless the reader of a message consciously chooses to decrypt it.

The shift cipher and the sufficient key space principle. Caesar's cipher suffers from the fact that encryption is always done in the same way, and there is no secret key. The shift cipher is similar to Caesar's cipher, but a secret key is introduced.[3] Specifically, in the shift cipher the key k is a number between 0 and 25. Then, to encrypt, letters are rotated by k places as in Caesar's cipher. Mapping this to the syntax of encryption described earlier, this means that algorithm Gen outputs a random number k in the set $\{0, \ldots, 25\}$; algorithm Enc takes a key k and a plaintext written using English letters and shifts each letter of the plaintext forward k positions (wrapping around from `z` to `a`); and algorithm Dec takes a key k and a ciphertext written using English letters and shifts every letter of the ciphertext *backward* k positions (this time wrapping around from `a` to `z`). The plaintext message space $\mathcal{M}$ is defined to be

[3] In some books, "Caesar's cipher" and "shift cipher" are used interchangeably.

all finite strings of characters from the English alphabet (note that numbers, punctuation, or other characters are not allowed in this scheme).

A more mathematical description of this method can be obtained by viewing the alphabet as the numbers $0, \ldots, 25$ (rather than as English characters). First, some notation: if a is an integer and N is an integer greater than 1, we define $[a \bmod N]$ as the remainder of a upon division by N. Note that $[a \bmod N]$ is an integer between 0 and $N-1$, inclusive. We refer to the process mapping a to $[a \bmod N]$ as *reduction modulo* N; we will have much more to say about reduction modulo N beginning in Chapter 7.

Using this notation, encryption of a plaintext character m_i with the key k gives the ciphertext character $[(m_i+k) \bmod 26]$, and decryption of a ciphertext character c_i is defined by $[(c_i-k) \bmod 26]$. In this view, the message space $\mathcal{M}$ is defined to be any finite sequence of integers that lie in the range $\{0, \ldots, 25\}$.

Is the shift cipher secure? Before reading on, try to decrypt the following message that was encrypted using the shift cipher and a secret key k (whose value we will not reveal):

`OVDTHUFWVZZPISLRLFZHYLAOLYL.`

Is it possible to decrypt this message without knowing k? Actually, it is completely trivial! The reason is that there are only 26 possible keys. Thus, it is easy to try every key, and see which key decrypts the ciphertext into a plaintext that "makes sense". Such an attack on an encryption scheme is called a *brute-force attack* or *exhaustive search*. Clearly, any secure encryption scheme must not be vulnerable to such a brute-force attack; otherwise, it can be completely broken, irrespective of how sophisticated the encryption algorithm is. This brings us to a trivial, yet important, principle called the "sufficient key space principle":

> *Any secure encryption scheme must have a key space that is not vulnerable to exhaustive search.*[4]

In today's age, an exhaustive search may use very powerful computers, or many thousands of PC's that are distributed around the world. Thus, the number of possible keys must be very large (at least 2^{60} or 2^{70}).

We emphasize that the above principle gives a *necessary* condition for security, not a *sufficient* one. We will see next an encryption scheme that has a very large key space but which is still insecure.

Mono-alphabetic substitution. The shift cipher maps each plaintext character to a different ciphertext character, but the mapping in each case is given by the same shift (the value of which is determined by the key). The idea

[4] This is actually only true if the message space is larger than the key space (see Chapter 2 for an example where security is achieved using a small key space as long as the message space is even smaller). In practice, when very long messages are typically encrypted with the same key, the key space must not be vulnerable to exhaustive search.

behind mono-alphabetic substitution is to map each plaintext character to a different ciphertext character in an *arbitrary* manner, subject only to the fact that the mapping must be one-to-one in order to enable decryption. The key space thus consists of all *permutations* of the alphabet, meaning that the size of the key space is $26! = 26 \cdot 25 \cdot 24 \cdots 2 \cdot 1$ (or approximately 2^{88}) if we are working with the English alphabet. As an example, the key

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
X	E	U	A	D	N	B	K	V	M	R	O	C	Q	F	S	Y	H	W	G	L	Z	I	J	P	T

in which `a` maps to `X`, etc., would encrypt the message `tellhimaboutme` to `GDOOKVCXEFLGCD`. A brute force attack on the key space for this cipher takes much longer than a lifetime, even using the most powerful computer known today. However, this does not necessarily mean that the cipher is secure. In fact, as we will show now, it is easy to break this scheme even though it has a very large key space.

Assume that English-language text is being encrypted (i.e., the text is grammatically-correct English writing, not just text written using characters of the English alphabet). It is then possible to attack the mono-alphabetic substitution cipher by utilizing statistical patterns of the English language (of course, the same attack works for any language). The two properties of this cipher that are utilized in the attack are as follows:

1. In this cipher, the mapping of each letter is fixed, and so if `e` is mapped to `D`, then every appearance of `e` in the plaintext will result in the appearance of `D` in the ciphertext.

2. The probability distribution of individual letters in the English language (or any other) is known. That is, the average frequency counts of the different English letters are quite invariant over different texts. Of course, the longer the text, the closer the frequency counts will be to the average. However, even relatively short texts (consisting of only tens of words) have distributions that are "close enough" to the average.

The attack works by tabulating the probability distribution of the ciphertext and then comparing it to the known probability distribution of letters in English text (see Figure 1.2). The probability distribution being tabulated in the attack is simply the frequency count of each letter in the ciphertext (i.e., a table saying that `A` appeared 4 times, `B` appeared 11 times, and so on). Then, we make an initial guess of the mapping defined by the key based on the frequency counts. For example, since `e` is the most frequent letter in English, we will guess that the most frequent character in the ciphertext corresponds to the plaintext character `e`, and so on. Unless the ciphertext is quite long, some of the guesses are likely to be wrong. Even for quite short ciphertexts, however, the guesses will be good enough to enable relatively quick decryption (especially utilizing other knowledge of the English language, such as the fact

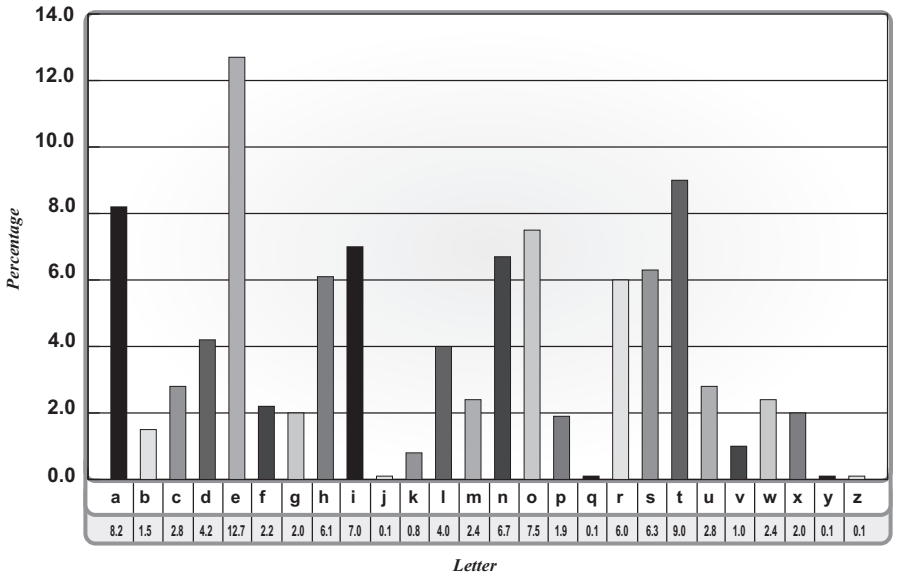

FIGURE 1.2: Average letter frequencies for English-language text.

that between `t` and `e`, the character `h` is likely to appear, and the fact that `u` generally follows `q`).

Actually, it should not be very surprising that the mono-alphabetic substitution cipher can be quickly broken, since puzzles based on this cipher appear in newspapers (and are solved by some people before their morning coffee)! We recommend that you try to decipher the following message — this should help convince you how easy the attack is to carry out (of course, you should use Figure 1.2 to help you):

```
JGRMQOYGHMVBJWRWQFPWHGFFDQGFPFZRKBEEBJIZQQOCIBZKLFAFGQVFZFWWE
OGWOPFGFHWOLPHLRLOLFDMFGQWBLWBWQOLKFWBYLBLYLFSFLJGRMQBOLWJVFP
FWQVHQWFFPQOQVFPQOCFPOGFWFJIGFQVHLHLROQVFGWJVFPFOLFHGQVQVFILE
OGQILHQFQGIQVVOSFAFGBWQVHQWIJVWJVFPFWHGFIWIHZZRQGBABHZQOCGFHX
```

We conclude that, although the mono-alphabetic cipher has a very large key space, it is still completely insecure.

An improved attack on the shift cipher. We can use character frequency tables to give an improved attack on the shift cipher. Specifically, our previous attack on the shift cipher required us to decrypt the ciphertext using each possible key, and then check to see which key results in a plaintext that "makes sense". A drawback of this approach is that it is difficult to automate, since it is difficult for a computer to check whether some plaintext "makes sense". (We do *not* claim this is impossible, as it can certainly be done using a dictionary of valid English words. We only claim that it is not trivial.) Moreover, there may be cases — we will see one below — where the plaintext characters are

distributed according to English-language text but the plaintext itself is not valid English text, making the problem harder.

As before, associate the letters of the English alphabet with the numbers $0, \ldots, 25$. Let p_i, for $0 \leq i \leq 25$, denote the probability of the ith letter in normal English text. A simple calculation using known values of p_i gives

$$\sum_{i=0}^{25} p_i^2 \approx 0.065\,. \tag{1.1}$$

Now, say we are given some ciphertext and let q_i denote the probability of the ith letter in this ciphertext (q_i is simply the number of occurrences of the ith letter divided by the length of the ciphertext). If the key is k, then we expect that q_{i+k} should be roughly equal to p_i for every i. (We use $i + k$ instead of the more cumbersome $[i + k \bmod 26]$.) Equivalently, if we compute

$$I_j \stackrel{\text{def}}{=} \sum_{i=0}^{25} p_i \cdot q_{i+j}$$

for each value of $j \in \{0, \ldots, 25\}$, then we expect to find that $I_k \approx 0.065$ where k is the key that is actually being used (whereas I_j for $j \neq k$ is expected to be different). This leads to a key-recovery attack that is easy to automate: compute I_j for all j, and then output the value k for which I_k is closest to 0.065.

The Vigenère (poly-alphabetic shift) cipher. As we have described, the statistical attack on the mono-alphabetic substitution cipher could be carried out because the mapping of each letter was fixed. Thus, such an attack can be thwarted by mapping different instances of the same plaintext character to different ciphertext characters. This has the effect of "smoothing out" the probability distribution of characters in the ciphertext. For example, consider the case that `e` is sometimes mapped to `G`, sometimes to `P`, and sometimes to `Y`. Then, the ciphertext letters `G`, `P`, and `Y` will most likely not stand out as more frequent, because other less-frequent characters will be also be mapped to them. Thus, counting the character frequencies will not offer much information about the mapping.

The Vigenère cipher works by applying multiple shift ciphers in sequence. That is, a short, secret word is chosen as the key, and then the plaintext is encrypted by "adding" each plaintext character to the next character of the key (as in the shift cipher), wrapping around in the key when necessary. For example, an encryption of the message `tellhimaboutme` using the key `cafe` would work as follows:

Plaintext:	`tellhimaboutme`
Key:	`cafecafecafeca`
Ciphertext:	`WFRQKJSFEPAYPF`

(The key need not be an actual English word.) This is exactly the same as encrypting the first, fifth, ninth, and so on characters with the shift cipher and key $k = 3$, the second, sixth, tenth, and so on characters with key $k = 1$, the third, seventh, and so on characters with $k = 6$ and the fourth, eighth, and so on characters with $k = 5$. Thus, it is a repeated shift cipher using different keys. Notice that in the above example `l` is mapped once to `R` and once to `Q`. Furthermore, the ciphertext character `F` is sometimes obtained from `e` and sometimes from `a`. Thus, the character frequencies in the ciphertext are "smoothed", as desired.

If the key is a sufficiently-long word (chosen at random), then cracking this cipher seems to be a daunting task. Indeed, it was considered by many to be an unbreakable cipher, and although it was invented in the 16th century a systematic attack on the scheme was only devised hundreds of years later.

Breaking the Vigenère cipher. A first observation in attacking the Vigenère cipher is that *if the length of the key is known*, then the task is relatively easy. Specifically, say the length of the key is t (this is sometimes called the period). Then the ciphertext can be divided into t parts where each part can be viewed as being encrypted using a single instance of the shift cipher. That is, let $k = k_1, \ldots, k_t$ be the key (each k_i is a letter of the alphabet) and let $c_1, c_2, \ldots$ be the ciphertext characters. Then, for every j $(1 \leq j \leq t)$ the set of characters

$$c_j, c_{j+t}, c_{j+2t}, \ldots$$

were all encrypted by a shift cipher using key k_j. All that remains is therefore to determine, for each j, which of the 26 possible keys is the correct one. This is not as trivial as in the case of the shift cipher, because by guessing a single letter of the key it is no longer possible to determine if the decryption "makes sense". Furthermore, checking for all values of j simultaneously would require a brute force search through 26^t different possible keys (which is infeasible for t greater than, say, 15). Nevertheless, we can still use the statistical method described earlier. That is, for every set of ciphertext characters relating to a given key (that is, for each value of j), it is possible to tabulate the frequency of each ciphertext character and then check which of the 26 possible shifts yields the "right" probability distribution. Since this can be carried out separately for each key, the attack can be carried out very quickly; all that is required is to build t frequency tables (one for each of the subsets of the characters) and compare them to the real probability distribution.

An alternate, somewhat easier approach, is to use the improved method for attacking the shift cipher that we showed earlier. Recall that this improved attack does not rely on checking for a plaintext that "makes sense", but only relies on the underlying probability distribution of characters in the plaintext.

Either of the above approaches give successful attacks when the key length is known. It remains to show how to determine the length of the key.

Kasiski's method, published in the mid-19th century, gives one approach for solving this problem. The first step is to identify repeated patterns of length 2

or 3 in the ciphertext. These are likely to be due to certain bigrams or trigrams that appear very often in the English language. For example, consider the word "`the`" that appears very often in English text. Clearly, "`the`" will be mapped to different ciphertext characters, depending on its position in the text. However, if it appears twice in the same relative position, then it will be mapped to the same ciphertext characters. For example, if it appears in positions $t + j$ and $2t + i$ (where $i \neq j$) then it will be mapped to different characters each time. However, if it appears in positions $t + j$ and $2t + j$, then it will be mapped to the same ciphertext characters. In a long enough text, there is a good chance that "`the`" will be mapped repeatedly to the same ciphertext characters.

Consider the following concrete example with the key `beads` (spaces have been added for clarity):

Plaintext:	`the man and the woman retrieved the letter from the post office`
Key:	`bea dsb ead sbe adsbe adsbeadsb ean sdeads bead sbe adsb eadbea`
Ciphertext:	`VMF QTP FOH MJJ XSFCS SIMTNFZXF YIS EIYUIK HWPQ MJJ QSLV TGJKGF`

The word `the` is mapped sometimes to `VMF`, sometimes to `MJJ` and sometimes to `YIS`. However, it is mapped *twice* to `MJJ`, and in a long enough text it is likely that it would be mapped multiple times to each of the possibilities. The main observation of Kasiski is that the distance between such multiple appearances (except for some coincidental ones) is a multiple of the period length. (In the above example, the period length is 5 and the distance between the two appearances of `MJJ` is 40, which is 8 times the period length.) Therefore, the *greatest common divisor* of all the distances between the repeated sequences should yield the period length t or a multiple thereof.

An alternative approach called the *index of coincidence method*, is a bit more algorithmic and hence easier to automate. Recall that if the key-length is t, then the ciphertext characters

$$c_1, c_{1+t}, c_{1+2t}, \ldots$$

are encrypted using the same shift. This means that the frequencies of the characters in this sequence are expected to be identical to the character frequencies of standard English text *except in some shifted order*. In more detail: let q_i denote the frequency of the ith English letter in the sequence above (once again, this is simply the number of occurrences of the ith letter divided by the total number of letters in the sequence). If the shift used here is k_1 (this is just the first character of the key), then we expect q_{i+k_1} to be roughly equal to p_i for all i, where p_i is again the frequency of the ith letter in standard English text. But this means that the sequence $p_0, \ldots, p_{25}$ is just the sequence $q_0, \ldots, q_{25}$ shifted by k_1 places. As a consequence, we expect that (see Equation (1.1)):

$$\sum_{i=0}^{25} q_i^2 = \sum_{i=0}^{25} p_i^2 \approx 0.065\,.$$

This leads to a nice way to determine the key length t. For $\tau = 1, 2, \ldots$, look at the sequence of ciphertext characters $c_1, c_{1+\tau}, c_{1+2\tau}, \ldots$ and tabulate $q_0, \ldots, q_{25}$ for this sequence. Then compute

$$S_\tau \stackrel{\text{def}}{=} \sum_{i=0}^{25} q_i^2.$$

When $\tau = t$ we expect to see $S_\tau \approx 0.065$ as discussed above. On the other hand, for $\tau \neq t$ we expect (roughly speaking) that all characters will occur with roughly equal probability in the sequence $c_1, c_{1+\tau}, c_{1+2\tau}, \ldots$, and so we expect $q_i \approx 1/26$ for all i. In this case we will obtain

$$S_\tau \approx \sum_{i=0}^{25} \frac{1}{26} \approx 0.038\,,$$

which is sufficiently different from 0.065 for this technique to work.

Ciphertext length and cryptanalytic attacks. The above attacks on the Vigenère cipher require a longer ciphertext than for previous schemes. For example, a large ciphertext is needed for determining the period if Kasiski's method is used. Furthermore, statistics are needed for t different parts of the ciphertext, and the frequency table of a message converges to the average as its length grows (and so the ciphertext needs to be approximately t times longer than in the case of the mono-alphabetic substitution cipher). Similarly, the attack that we showed for the mono-alphabetic substitution cipher requires a longer ciphertext than for the attacks on the shift cipher (which can work for messages consisting of just a single word). This phenomenon is not coincidental, and relates to the size of the key space for each encryption scheme.

Ciphertext-only vs. known-plaintext attacks. The attacks described above are all ciphertext-only attacks (recall that this is the easiest type of attack to carry out in practice). All the above ciphers are *trivially broken* if the adversary is able to carry out a known-plaintext attack; we leave a demonstration of this as an exercise.

Conclusions and discussion. We have presented only a few historical ciphers. Beyond their general historical interest, our aim in presenting them was to illustrate some important lessons regarding cryptographic design. Stated briefly, these lessons are:

1. *Sufficient key space principle:* Assuming sufficiently-long messages are being encrypted, a secure encryption scheme must have a key space that cannot be searched exhaustively in a reasonable amount of time. However, a large key space does not by itself imply security (e.g., the mono-alphabetic substitution cipher has a large key space but is trivial to break). Thus, a large key space is a necessary requirement, but not a sufficient one.

2. *Designing secure ciphers is a hard task:* The Vigenère cipher remained unbroken for a long time, partially due to its presumed complexity. Far more complex schemes have also been used, such as the German Enigma. Nevertheless, this complexity does not imply security and all historical ciphers can be completely broken. In general, it is very hard to design a secure encryption scheme, and such design should be left to experts.

The history of classical encryption schemes is fascinating, both with respect to the methods used as well as the influence of cryptography and cryptanalysis on world history (in World War II, for example). Here, we have only tried to give a taste of some of the more basic methods, with a focus on what modern cryptography can learn from these attempts.

1.4 The Basic Principles of Modern Cryptography

The previous section has given a taste of historical cryptography. It is fair to say that, historically, cryptography was more of an art than any sort of science: schemes were designed in an ad-hoc manner and then evaluated based on their perceived complexity or cleverness. Unfortunately, as we have seen, all such schemes (no matter how clever) were eventually broken.

Modern cryptography, now resting on firmer and more scientific foundations, gives hope of breaking out of the endless cycle of constructing schemes and watching them get broken. In this section we outline the main principles and paradigms that distinguish modern cryptography from classical cryptography. We identify three main principles:

1. *Principle 1* — the first step in solving any cryptographic problem is the formulation of a rigorous and precise definition of security.

2. *Principle 2* — when the security of a cryptographic construction relies on an unproven assumption, this assumption must be precisely stated. Furthermore, the assumption should be as minimal as possible.

3. *Principle 3* — cryptographic constructions should be accompanied by a rigorous proof of security with respect to a definition formulated according to principle 1, and relative to an assumption stated as in principle 2 (if an assumption is needed at all).

We now discuss each of these principles in greater depth.

1.4.1 Principle 1 – Formulation of Exact Definitions

One of the key intellectual contributions of modern cryptography has been the realization that formal definitions of security are *essential* prerequisites

for the design, usage, or study of any cryptographic primitive or protocol. Let us explain each of these in turn:

1. *Importance for design:* Say we are interested in constructing a secure encryption scheme. If we do not have a firm understanding of what it is we want to achieve, how can we possibly know whether (or when) we have achieved it? Having an exact definition in mind enables us to better direct our design efforts, as well as to evaluate the quality of what we build, thereby improving the end construction. In particular, it is much better to define what is needed *first* and then begin the design phase, rather than to come up with a *post facto* definition of what has been achieved once the design is complete. The latter approach risks having the design phase end when the designers' patience is tried (rather than when the goal has been met), or may result in a construction that achieves *more* than is needed and is thus less efficient than a better solution.

2. *Importance for usage:* Say we want to use an encryption scheme within some larger system. How do we know which encryption scheme to use? If presented with a candidate encryption scheme, how can we tell whether it suffices for our application? Having a precise definition of the security achieved by a given scheme (coupled with a security proof relative to a formally-stated assumption as discussed in principles 2 and 3) allows us to answer these questions. Specifically, we can define the security that *we* desire in our system (see point 1, above), and then verify whether the definition satisfied by a given encryption scheme suffices for our purposes. Alternatively, we can specify the definition that we need the encryption scheme to satisfy, and look for an encryption scheme satisfying this definition. Note that it may not be wise to choose the "most secure" scheme, since a weaker notion of security may suffice for our application and we may then be able to use a more efficient scheme.

3. *Importance for study:* Given two encryption schemes, how can we compare them? Without any definition of security, the only point of comparison is efficiency, but efficiency alone is a poor criterion since a highly efficient scheme that is completely insecure is of no use. Precise specification of the level of security achieved by a scheme offers another point of comparison. If two schemes are equally efficient but the first one satisfies a stronger definition of security than the second, then the first is preferable.[5] There may also be a trade-off between security and efficiency (see the previous two points), but at least with precise definitions we can understand what this trade-off entails.

[5] Of course, things are rarely this simple.

Of course, precise definitions also enable rigorous proofs (as we will discuss when we come to principle 3), but the above reasons stand irrespective of this.

It is a mistake to think that formal definitions are not needed since "we have an intuitive idea of what security means". For starters, different people have different intuition regarding what is considered secure. Even one person might have multiple intuitive ideas of what security means, depending on the context. For example, in Chapter 3 we will study four different definitions of security for private-key encryption, each of which is useful in a different scenario. In any case, a formal definition is necessary for communicating your "intuitive idea" to someone else.

An example: secure encryption. It is also a mistake to think that formalizing definitions is trivial. For example, how would you formalize the desired notion of security for private-key encryption? (The reader may want to pause to think about this before reading on.) We have asked students many times how secure encryption should be defined, and have received the following answers (often in the following order):

1. *Answer 1 — an encryption scheme is secure if no adversary can find the secret key when given a ciphertext.* Such a definition of encryption completely misses the point. The aim of encryption is to protect the message being encrypted and the secret key is just the means of achieving this. To take this to an absurd level, consider an encryption scheme that ignores the secret key and just outputs the plaintext. Clearly, no adversary can find the secret key. However, it is also clear that no secrecy whatsoever is provided.[6]

2. *Answer 2 — an encryption scheme is secure if no adversary can find the plaintext that corresponds to the ciphertext.* This definition already looks better and can even be found in some texts on cryptography. However, after some more thought, it is also far from satisfactory. For example, an encryption scheme that reveals 90% of the plaintext would still be considered secure under this definition, as long as it is hard to find the remaining 10%. But this is clearly unacceptable in most common applications of encryption. For example, employment contracts are mostly standard text, and only the salary might need to be kept secret; if the salary is in the 90% of the plaintext that is revealed then nothing is gained by encrypting.

   If you find the above counterexample silly, refer again to footnote 6. The point once again is that if the definition as stated isn't what was meant, then a scheme could be proven secure without actually providing the necessary level of protection. (This is a good example of why *exact* definitions are important.)

[6] And lest you respond: "But that's not what I meant!", well, that's exactly the point: it is often not so trivial to formalize what one means.

3. *Answer 3 — an encryption scheme is secure if no adversary can determine any character of the plaintext that corresponds to the ciphertext.* This already looks like an excellent definition. However, other subtleties can arise. Going back to the example of the employment contract, it may be impossible to determine the actual salary or even any digit thereof. However, should the encryption scheme be considered secure if it leaks whether the encrypted salary is greater than or less than $100,000 per year? Clearly not. This leads us to the next suggestion.

4. *Answer 4 — an encryption scheme is secure if no adversary can derive any meaningful information about the plaintext from the ciphertext.* This is already close to the actual definition. However, it is lacking in one respect: it does not define what it means for information to be "meaningful". Different information may be meaningful in different applications. This leads to a very important principle regarding definitions of security for cryptographic primitives: *definitions of security should suffice for all potential applications.* This is essential because one can never know what applications may arise in the future. Furthermore, implementations typically become part of general cryptographic libraries which are then used in may different contexts and for many different applications. Security should ideally be guaranteed for all possible uses.

5. *The final answer — an encryption scheme is secure if no adversary can compute* any *function of the plaintext from the ciphertext.* This provides a very strong guarantee and, when formulated properly, is considered today to be the "right" definition of security for encryption. Even here, there are questions regarding the attack model that should be considered, and how this aspect of security should be defined.

Even though we have now hit upon the correct requirement for secure encryption, conceptually speaking, it remains to state this requirement mathematically and formally, and this is in itself a non-trivial task (one that we will address in detail in Chapters 2 and 3).

As noted in the "final answer", above, our formal definition must also specify the attack model: i.e., whether we assume a ciphertext-only attack or a chosen-plaintext attack. This illustrates a general principle used when formulating cryptographic definitions. Specifically, in order to fully define security of some cryptographic task, there are two distinct issues that must be explicitly addressed. The first is what is considered to be a break, and the second is what is assumed regarding the power of the adversary. The break is exactly what we have discussed above; i.e., an encryption scheme is considered broken if an adversary learns some function of the plaintext from a ciphertext. The *power of the adversary* relates to assumptions regarding the actions the adversary is assumed to be able to take, as well as the adversary's computational power. The former refers to considerations such as whether the adversary is assumed only to be able to eavesdrop on encrypted messages

(i.e., a ciphertext-only attack), or whether we assume that the adversary can also actively request encryptions of any plaintext that it likes (i.e., carry out a chosen-plaintext attack). A second issue that must be considered is the computational power of the adversary. For all of this book, except Chapter 2, we will want to ensure security against any *efficient* adversary, by which we mean any adversary running in polynomial time. (A full discussion of this point appears in Section 3.1.2. For now, it suffices to say that an "efficient" strategy is one that can be carried out in a lifetime. Thus "feasible" is arguably a more accurate term.) When translating this into concrete terms, we might require security against any adversary utilizing decades of computing time on a supercomputer.

In summary, any definition of security will take the following general form:

> *A cryptographic scheme for a given task is secure if no adversary of a specified power can achieve a specified break.*

We stress that the definition never assumes anything about the adversary's *strategy*. This is an important distinction: we are willing to assume something about the adversary's capabilities (e.g., that it is able to mount a chosen-plaintext attack but not a chosen-ciphertext attack), but we are *not* willing to assume anything about *how it uses* its abilities. We call this the "arbitrary adversary principle": security must be guaranteed for *any* adversary within the class of adversaries having the specified power. This principle is important because it is impossible to foresee what strategies might be used in an adversarial attack (and history has proven that attempts to do so are doomed to failure).

Mathematics and the real world. A definition of security essentially provides a mathematical formulation of a real-world problem. If the mathematical definition does not appropriately model the real world, then the definition may be useless. For example, if the adversarial power under consideration is too weak (and, in practice, adversaries have more power), or the break is such that it allows real attacks that were not foreseen (like one of the early answers regarding encryption), then "real security" is not obtained, even if a "mathematically-secure" construction is used. In short, a definition of security must accurately model the real world in order for it to deliver on its mathematical promise of security.

It is quite common, in fact, for a widely-accepted definition to be ill-suited for some new application. As one notable example, there are encryption schemes that were proven secure (relative to some definition like the ones we have discussed above) and then implemented on smart-cards. Due to physical properties of the smart-cards, it was possible for an adversary to monitor the *power usage* of the smart-card (e.g., how this power usage fluctuated over time) as the encryption scheme was being run, and it turned out that this information could be used to determine the key. There was nothing wrong with the security definition or the proof that the scheme satisfied this

definition; the problem was simply that there was a mismatch between the definition and the real-world implementation of the scheme on a smart-card.

This should not be taken to mean that definitions (or proofs, for that matter) are useless! The definition — and the scheme that satisfies it — may still be appropriate for other settings, such as when encryption is performed on an end-host whose power usage cannot be monitored by an adversary. Furthermore, one way to achieve secure encryption on a smart-card would be to further *refine* the definition so that it takes power analysis into account. Or, perhaps hardware countermeasures for power analysis can be developed, with the effect of making the original definition (and hence the original scheme) appropriate for smart-cards. The point is that with a definition you at least know where you stand, even if the definition turns out not to accurately model the particular setting in which a scheme is used. In contrast, with no definition it is not even clear what went wrong.

This possibility of a disconnect between a mathematical model and the reality it is supposed to be modeling is not unique to cryptography but is something that occurs throughout science. To take an example from the field of computer science, consider the meaning of a *mathematical proof* that there exist well-defined problems that computers cannot solve.[7] The immediate question that arises is *what does it mean for "a computer to solve a problem"*? Specifically, a mathematical proof can be provided only when there is some mathematical definition of what a computer is (or to be more exact, what the process of computation is). The problem is that computation is a real-world process, and there are many different ways of computing. In order for us to be really convinced that the "unsolvable problem" is really unsolvable, we must be convinced that our mathematical definition of computation captures the *real-world process* of computation. How do we know when it does?

This inherent difficulty was noted by Alan Turing who studied questions of what can and cannot be solved by a computer. We quote from his original paper [140] (the text in square brackets replaces original text in order to make it more reader friendly):

> *No attempt has yet been made to show [that the problems we have defined to be solvable by a computer] include [exactly those problems] which would naturally be regarded as computable. All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue is "What are the possible processes which can be carried out in [computation]?"*
>
> *The arguments which I shall use are of three kinds.*
>
> (a) *A direct appeal to intuition.*

[7] Those who have taken a course in computability theory will be familiar with the fact that such problems do indeed exist (e.g., the Halting Problem).

(b) *A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).*

(c) *Giving examples of large classes of [problems that can be solved using a given definition of computation].*

In some sense, Turing faced the exact same problem as cryptographers. He developed a mathematical model of computation but needed to somehow be convinced that the model was a good one. Likewise, cryptographers define notions of security and need to be convinced that their definitions imply meaningful security guarantees in the real world. As with Turing, they may employ the following tools to become convinced:

1. *Appeals to intuition:* the first tool when contemplating a new definition of security is to see whether it implies security properties that we intuitively expect to hold. This is a minimum requirement, since (as we have seen in our discussion of encryption) our initial intuition usually results in a notion of security that is too weak.

2. *Proofs of equivalence:* it is often the case that a new definition of security is justified by showing that it is equivalent to (or stronger than) a definition that is older, more familiar, or more intuitively-appealing.

3. *Examples:* a useful way of being convinced that a definition of security suffices is to show that the different real-world attacks we are familiar with are ruled out by the definition.

In addition to all of the above, and perhaps most importantly, we rely on the *test of time* and the fact that with time, the scrutiny and investigation of both researchers and practitioners testifies to the soundness of a definition.

1.4.2 Principle 2 – Reliance on Precise Assumptions

Most modern cryptographic constructions cannot be proven secure unconditionally. Indeed, proofs of this sort would require resolving questions in the theory of computational complexity that seem far from being answered today. The result of this unfortunate state of affairs is that security typically relies upon some assumption. The second principle of modern cryptography states that assumptions must be precisely stated. This is for three main reasons:

1. *Validation of the assumption:* By their very nature, assumptions are statements that are not proven but are rather conjectured to be true. In order to strengthen our belief in some assumption, it is necessary for the assumption to be studied. The more the assumption is examined and tested without being successfully refuted, the more confident we are that the assumption is true. Furthermore, study of an assumption can provide positive evidence of its validity by showing that it is implied by some other assumption that is also widely believed.

If the assumption being relied upon is not precisely stated and presented, it cannot be studied and (potentially) refuted. Thus, a pre-condition to raising our confidence in an assumption is having a precise statement of what exactly is assumed.

2. *Comparison of schemes:* Often in cryptography, we may be presented with two schemes that can both be proven to satisfy some definition but each with respect to a different assumption. Assuming both schemes are equally efficient, which scheme should be preferred? If the assumption on which one scheme is based is *weaker* than the assumption on which the second scheme is based (i.e., the second assumption implies the first), then the first scheme is to be preferred since it may turn out that the second assumption is false while the first assumption is true. If the assumptions used by the two schemes are incomparable, then the general rule is to prefer the scheme that is based on the better-studied assumption, or the assumption that is simpler (for the reasons highlighted in the previous paragraphs).

3. *Facilitation of proofs of security:* As we have stated, and will discuss in more depth in principle 3, modern cryptographic constructions are presented together with proofs of security. If the security of the scheme cannot be proven unconditionally and must rely on some assumption, then a mathematical proof that "the construction is secure if the assumption is true" can only be provided if there is a precise statement of what the assumption is.

One observation is that it is always possible to just assume that a construction *itself* is secure. If security is well defined, this is also a precise assumption (and the proof of security for the construction is trivial)! Of course, this is not accepted practice in cryptography for a number of reasons. First of all, as noted above, an assumption that has been tested over the years is preferable to a new assumption that is introduced just to prove a given construction secure. Second, there is a general preference for assumptions that are simpler to state, since such assumptions are easier to study and to refute. So, for example, an assumption of the type that some mathematical problem is hard to solve is simpler to study and work with than an assumption that an encryption schemes satisfies a complex (and possibly unnatural) security definition. When a simple assumption is studied at length and still no refutation is found, we have greater confidence in its being correct. Another advantage of relying on "lower-level" assumptions (rather than just assuming a construction is secure) is that these low-level assumptions can typically be shared amongst a number of constructions. If a specific instantiation of the assumption turns out to be false, it can simply be replaced (within any higher-level construction based on that assumption) by a *different* instantiation of that assumption.

The above methodology is used throughout this book. For example, Chapters 3 and 4 show how to achieve secure communication (in a number of ways),

assuming that a primitive called a "pseudorandom function" exists. In these chapters nothing is said at all about how such a primitive can be constructed. In Chapter 5, we then discuss how pseudorandom functions are constructed in practice, and in Chapter 6 we show that pseudorandom functions can be constructed from even lower-level primitives.

1.4.3 Principle 3 – Rigorous Proofs of Security

The first two principles discussed above lead naturally to the current one. Modern cryptography stresses the importance of rigorous proofs of security for proposed schemes. The fact that exact definitions and precise assumptions are used means that such a proof of security is possible. However, why is a proof necessary? The main reason is that the security of a construction or protocol cannot be checked in the same way that software is typically checked. For example, the fact that encryption and decryption "work" and that the ciphertext looks garbled, does not mean that a sophisticated adversary is unable to break the scheme. Without a *proof* that no adversary of the specified power can break the scheme, we are left only with our intuition that this is the case. Experience has shown that intuition in cryptography and computer security is disastrous. There are countless examples of unproven schemes that were broken, sometimes immediately and sometimes years after being presented or deployed.

Another reason why proofs of security are so important is related to the potential damage that can result if an insecure system is used. Although software bugs can sometimes be very costly, the potential damage that may result from someone breaking the encryption scheme or authentication mechanism of a bank is huge. Finally, we note that although many bugs exist in software, things basically work due to the fact that typical users do not try to make their software fail. In contrast, attackers use amazingly complex and intricate means (utilizing specific properties of the construction) to attack security mechanisms with the clear aim of breaking them. Thus, although proofs of correctness are always desirable in computer science, they are absolutely essential in the realm of cryptography and computer security. We stress that the above observations are not just hypothetical, but are conclusions that have been reached after years of empirical evidence and experience.

The reductionist approach. We conclude by noting that most proofs in modern cryptography use what may be called the reductionist approach. Given a theorem of the form

> *"Given that Assumption X is true, Construction Y is secure according to the given definition",*

a proof typically shows how to *reduce* the problem given by Assumption X to the problem of breaking Construction Y. More to the point, the proof will typically show (via a constructive argument) how any adversary breaking

Construction Y can be used as a sub-routine to violate Assumption X. We will have more to say about this in Section 3.1.3.

Summary – Rigorous vs. Ad-Hoc Approaches to Security

The combination of the above three principles constitutes a rigorous approach to cryptography that is distinct from the ad-hoc approach of classical cryptography. The ad-hoc approach may fail on any one of the above three principles, but often ignores them all. Unfortunately, ad hoc solutions are still designed and deployed by those who wish to obtain a "quick and dirty" solution to a problem (or by those who are just simply unaware). We hope that this book will contribute to an awareness of the importance of the rigorous approach, and its success in developing new, mathematically-secure schemes.

References and Additional Reading

In this chapter, we have studied just a few of the known historical ciphers. There are many others of both historical and mathematical interest, and we refer the reader to textbooks by Stinson [138] or Trappe and Washington [139] for further details. The role of these schemes in history (and specifically in the history of war) is a fascinating subject that is covered in the book by Kahn [81].

We discussed the differences between the historical, non-rigorous approach to cryptography (as exemplified by historical ciphers) and a rigorous approach based on precise definitions and proofs. Shannon [127] was the first to take the latter approach. Modern cryptography, which relies on (computational) assumptions in addition to definitions and proofs, was begun in the seminal paper by Goldwasser and Micali [69]. We will study this in Chapter 3. A comprehensive coverage of the modern cryptographic approach can be found in Goldreich's work on the Foundations of Cryptography [64, 65]. Our presentation in a number of places was influenced by this work, most notably in Chapter 6.

Exercises

1.1 Decrypt the ciphertext provided at the end of the section on mono-alphabetic substitution.

1.2 Provide a formal definition of the Gen, Enc, and Dec algorithms for both the mono-alphabetic substitution and Vigenère ciphers.

1.3 Consider an improved version of the Vigenère cipher, where instead of using multiple shift ciphers, multiple mono-alphabetic substitution ciphers are used. That is, the key consists of t random permutations of the alphabet, and the plaintext characters in positions $i, t+i, 2t+i$, and so on are encrypted using the ith permutation. Show how to break this version of the cipher.

1.4 In an attempt to prevent Kasiski's attack on the Vigenère cipher, the following modification has been proposed. Given the period t of the cipher, the plaintext is broken up into blocks of size t. Recall that within each block, the Vigenère cipher works by encrypting the ith character with the ith key (using a shift cipher). Letting the key be $k_1, \ldots, k_t$, this means the ith character in each block is encrypted by adding k_i to it, modulo 26. The proposed modification is to encrypt the ith character in the jth block by adding $k_i + j$ modulo 26.

(a) Show that decryption can be carried out.

(b) Describe the effect of the above modification on Kasiski's attack.

(c) Devise an alternate way to determine the period for this scheme.

1.5 Show that the shift, substitution, and Vigenère ciphers are all trivial to break using a known-plaintext attack. How much known plaintext is needed to completely recover the key for each of the ciphers?

1.6 Show that the shift, substitution, and Vigenère ciphers are all trivial to break using a chosen-plaintext attack. How much plaintext must be encrypted in order for the adversary to completely recover the key? Compare to the previous question.

Chapter 2

Perfectly-Secret Encryption

In the previous chapter, we presented historical encryption schemes (ciphers) and showed how they can be completely broken with very little computational effort. In this chapter, we look at the other extreme and study encryption schemes that are *provably secure* even against an adversary who has unbounded computational power. Such schemes are called *perfectly secret*. We will see under what conditions perfect secrecy can and cannot be achieved, and why this is the case.

The material in this chapter belongs, in some sense, more to the world of "classical cryptography" than to the world of "modern cryptography". Besides the fact that all the material introduced here was developed before the revolution in cryptography that took place in the mid-'70s and '80s, the constructions we study in this chapter rely only on the first and third principles outlined in Section 1.4. That is, precise mathematical definitions will be given and rigorous proofs will be shown, but it will not be necessary to rely on any unproven assumptions. This is clearly advantageous. We will see, however, that such an approach has inherent limitations. Thus, in addition to serving as a good basis for understanding the principles underlying modern cryptography, the results of this chapter also justify our later adoption of all three of the aforementioned principles.

In this chapter, we assume a familiarity with basic probability. The relevant notions are reviewed in Appendix A.3.

2.1 Definitions and Basic Properties

We begin by briefly recalling some of the syntax that was introduced in the previous chapter. An encryption scheme is defined by three algorithms Gen, Enc, and Dec, as well as a specification of a message space $\mathcal{M}$ with $|\mathcal{M}| > 1$.[1] The key-generation algorithm Gen is a probabilistic algorithm that outputs a key k chosen according to some distribution. We denote by $\mathcal{K}$ the key space, i.e., the set of all possible keys that can be output by Gen, and

[1] If $|\mathcal{M}| = 1$ there is only one message and there is no point in communicating, let alone encrypting.

require $\mathcal{K}$ to be finite. The encryption algorithm Enc takes as input a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, and outputs a ciphertext c; we denote this by $\mathsf{Enc}_k(m)$. The encryption algorithm may be probabilistic, so that $\mathsf{Enc}_k(m)$ might output a different ciphertext when run multiple times. To emphasize this, we write $c \leftarrow \mathsf{Enc}_k(m)$ to denote the possibly probabilistic process by which message m is encrypted using key k to give ciphertext c. (In case Enc is deterministic, we may emphasize this by writing $c := \mathsf{Enc}_k(m)$.) We let $\mathcal{C}$ denote the set of all possible ciphertexts that can be output by $\mathsf{Enc}_k(m)$, for all possible choices of $k \in \mathcal{K}$ and $m \in \mathcal{M}$ (and for all random choices of Enc in case it is randomized). The decryption algorithm Dec takes as input a key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ and outputs a message $m \in \mathcal{M}$. Throughout the book, we assume that encryption schemes are *perfectly correct*, meaning that for all $k \in \mathcal{K}$, $m \in \mathcal{M}$, and any ciphertext c output by $\mathsf{Enc}_k(m)$, it holds that $\mathsf{Dec}_k(c) = m$ with probability 1. This implies that we may assume Dec is deterministic without loss of generality (since $\mathsf{Dec}_k(c)$ must give the same output every time it is run). We will thus write $m := \mathsf{Dec}_k(c)$ to denote the process of decrypting ciphertext c using key k.

In the definitions and theorems below, we refer to probability distributions over $\mathcal{K}$, $\mathcal{M}$, and $\mathcal{C}$. The distribution over $\mathcal{K}$ is simply the one that is defined by running Gen and taking the output. (As noted previously, it is almost always the case that Gen chooses a key uniformly from $\mathcal{K}$; moreover, this may be assumed to be the case without loss of generality.) For $k \in \mathcal{K}$, we let $\Pr[K = k]$ denote the probability that the key output by Gen is equal to k. (Formally, K is a random variable denoting the value of the key.) Similarly, for $m \in \mathcal{M}$ we let $\Pr[M = m]$ denote the probability that the message is equal to m. The fact that the message is chosen according to some distribution (rather than being fixed) is meant to model the fact that, at least from the point of view of the adversary, different messages have different probabilities of being sent. (If the adversary knows what message is being sent, then it doesn't need to decrypt anything and there is no need for the parties to use encryption!) As an example, the adversary may know that the encrypted message is either `attack tomorrow` or `don't attack`. Furthermore, the adversary may even know (by other means) that with probability 0.7 the message will be a command to attack and with probability 0.3 the message will be a command not to attack. In this case, we have $\Pr[M = \texttt{attack tomorrow}] = 0.7$ and $\Pr[M = \texttt{don't attack}] = 0.3$.

The distributions over $\mathcal{K}$ and $\mathcal{M}$ are independent, i.e., the key and message are chosen independently. This is the case because the key is chosen and fixed (i.e., shared by the communicating parties) before the message is known. Furthermore, the distribution over $\mathcal{K}$ is fixed by the encryption scheme itself (since it is defined by Gen) while the distribution over $\mathcal{M}$ may vary depending on the parties who are using the encryption scheme.

For $c \in \mathcal{C}$, we write $\Pr[C = c]$ to denote the probability that the ciphertext is c. Given the encryption algorithm Enc, the distribution over $\mathcal{C}$ is fully determined by the distributions over $\mathcal{K}$ and $\mathcal{M}$.

The definition. We are now ready to define the notion of perfect secrecy. Intuitively, we imagine an adversary who knows the probability distribution over $\mathcal{M}$; that is, the adversary knows the likelihood that different messages will be sent (as in the example given above). The adversary then observes some ciphertext being sent by one party to the other. Ideally, observing this ciphertext should have *no effect* on the knowledge of the adversary; in other words, the *a posteriori* likelihood that some message m was sent (even given the ciphertext that was seen) should be no different from the *a priori* probability that m would be sent. This should hold for any $m \in \mathcal{M}$. Furthermore, this should hold even if the adversary has unbounded computational power. This means that a ciphertext reveals nothing about the underlying plaintext, and thus an adversary who intercepts a ciphertext learns absolutely nothing about the plaintext that was encrypted. Formally:

DEFINITION 2.1 *An encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *over a message space* $\mathcal{M}$ *is* perfectly secret *if for every probability distribution over* $\mathcal{M}$*, every message* $m \in \mathcal{M}$*, and every ciphertext* $c \in \mathcal{C}$ *for which* $\Pr[C = c] > 0$*:*

$$\Pr[M = m \mid C = c] = \Pr[M = m].$$

(The requirement that $\Pr[C = c] > 0$ is a technical one needed to prevent conditioning on a zero-probability event.) Another way of interpreting Definition 2.1 is that a scheme is perfectly secret if the distributions over messages and ciphertexts are *independent*.

A simplifying convention. In this chapter, we consider only probability distributions over $\mathcal{M}$ and $\mathcal{C}$ that assign *non-zero probabilities* to all $m \in \mathcal{M}$ and $c \in \mathcal{C}$.[2] This significantly simplifies the presentation because we often need to divide by $\Pr[M = m]$ or $\Pr[C = c]$, which is a problem if they may equal zero. Likewise, as in Definition 2.1 we sometimes need to condition on the event $C = c$ or $M = m$. This too is problematic if those events have zero probability. Due to this simplifying convention, we will not mention from here on the requirement that $\Pr[M = m] > 0$ or $\Pr[C = c] > 0$, even though it is actually required.

We stress that this convention is only meant to simplify the exposition and is not a fundamental limitation. In particular all the theorems we prove can be appropriately adapted to the case of arbitrary distributions over $\mathcal{M}$ and $\mathcal{C}$ (that may assign some messages or ciphertexts probability 0). See also Exercise 2.6.

An equivalent formulation. The following lemma gives an equivalent formulation of Definition 2.1.

[2] We remark that this holds always for $k \in \mathcal{K}$ because $\mathcal{K}$ is defined as the set of keys output by Gen with non-zero probability.

LEMMA 2.2 *An encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *over a message space* $\mathcal{M}$ *is perfectly secret if and only if for every probability distribution over* $\mathcal{M}$*, every message* $m \in \mathcal{M}$*, and every ciphertext* $c \in \mathcal{C}$*:*

$$\Pr[C = c \mid M = m] = \Pr[C = c].$$

PROOF Fix a distribution over $\mathcal{M}$ and arbitrary $m \in \mathcal{M}$ and $c \in \mathcal{C}$. Say

$$\Pr[C = c \mid M = m] = \Pr[C = c].$$

Multiplying both sides of the equation by $\Pr[M = m]/\Pr[C = c]$ gives

$$\frac{\Pr[C = c \mid M = m] \cdot \Pr[M = m]}{\Pr[C = c]} = \Pr[M = m].$$

Using Bayes' theorem (see Theorem A.8), the left-hand-side is exactly equal to $\Pr[M = m \mid C = c]$. Thus, $\Pr[M = m \mid C = c] = \Pr[M = m]$ and the scheme is perfectly secret.

The other direction of the proof is left as an exercise. ■

We emphasize that in the above proof we used the fact that (by the simplifying convention mentioned earlier) both $m \in \mathcal{M}$ and $c \in \mathcal{C}$ are assigned non-zero probabilities, enabling division by $\Pr[C = c]$ and conditioning on the event $M = m$.

Perfect indistinguishability. We now use Lemma 2.2 to obtain another equivalent and useful formulation of perfect secrecy. This formulation states that the probability distribution over $\mathcal{C}$ is independent of the plaintext. That is, let $\mathcal{C}(m)$ denote the distribution of the ciphertext when the message being encrypted is $m \in \mathcal{M}$ (this distribution depends on the choice of key, as well as the randomness of the encryption algorithm in case it is probabilistic). Then the claim is that for every $m_0, m_1 \in \mathcal{M}$, the distributions $\mathcal{C}(m_0)$ and $\mathcal{C}(m_1)$ are identical. This is just another way of saying that the ciphertext contains no information about the plaintext. We refer to this formulation as *perfect indistinguishability* because it implies that it is impossible to distinguish an encryption of m_0 from an encryption of m_1 (due to the fact that the distribution over the ciphertext is the same in each case).

LEMMA 2.3 *An encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *over a message space* $\mathcal{M}$ *is perfectly secret if and only if for every probability distribution over* $\mathcal{M}$*, every* $m_0, m_1 \in \mathcal{M}$*, and every* $c \in \mathcal{C}$*:*

$$\Pr[C = c \mid M = m_0] = \Pr[C = c \mid M = m_1].$$

PROOF Assume that the encryption scheme is perfectly secret and fix messages $m_0, m_1 \in \mathcal{M}$ and a ciphertext $c \in \mathcal{C}$. By Lemma 2.2 we have

$$\Pr[C = c \mid M = m_0] = \Pr[C = c] = \Pr[C = c \mid M = m_1],$$

completing the proof of the first direction.

Assume next that for every distribution over $\mathcal{M}$, every $m_0, m_1 \in \mathcal{M}$, and every $c \in \mathcal{C}$ it holds that $\Pr[C = c \mid M = m_0] = \Pr[C = c \mid M = m_1]$. Fix some distribution over $\mathcal{M}$, and an arbitrary $m_0 \in \mathcal{M}$ and $c \in \mathcal{C}$. Define $p \stackrel{\text{def}}{=} \Pr[C = c \mid M = m_0]$. Since $\Pr[C = c \mid M = m] = \Pr[C = c \mid M = m_0] = p$ for all m, we have

$$\begin{aligned}
\Pr[C = c] &= \sum_{m \in \mathcal{M}} \Pr[C = c \mid M = m] \cdot \Pr[M = m] \\
&= \sum_{m \in \mathcal{M}} p \cdot \Pr[M = m] \\
&= p \cdot \sum_{m \in \mathcal{M}} \Pr[M = m] \\
&= p \\
&= \Pr[C = c \mid M = m_0].
\end{aligned}$$

Since m_0 was arbitrary, we have shown that $\Pr[C = c] = \Pr[C = c \mid M = m]$ for all $c \in \mathcal{C}$ and $m \in \mathcal{M}$. Applying Lemma 2.2, we conclude that the encryption scheme is perfectly secret. ■

Adversarial indistinguishability. We conclude this section by presenting another equivalent definition of perfect secrecy. This definition is based on an *experiment* involving an adversary $\mathcal{A}$, and formalizes $\mathcal{A}$'s inability to distinguish the encryption of one plaintext from the encryption of another; we thus call it *adversarial indistinguishability*. This definition will serve as our starting point when we introduce the notion of computational security in the next chapter. Throughout the book we will often use experiments in order to define security. These "experiments" are essentially a game played between an adversary trying to break a cryptographic scheme and an imaginary tester who wishes to see if the adversary succeeds.

We define an experiment that we call $\mathsf{PrivK}^{\mathsf{eav}}$ since it considers the setting of private-key encryption and an eavesdropping adversary (the adversary is eavesdropping because it only receives a ciphertext c and then tries to determine something about the plaintext). The experiment is defined for any encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over message space $\mathcal{M}$ and for any adversary $\mathcal{A}$. We let $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$ denote an execution of the experiment for a given Π and $\mathcal{A}$. The experiment is defined as follows:

The eavesdropping indistinguishability experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$:

1. *The adversary $\mathcal{A}$ outputs a pair of messages $m_0, m_1 \in \mathcal{M}$.*
2. *A random key k is generated by running Gen, and a random bit $b \leftarrow \{0,1\}$ is chosen. (These are chosen by some imaginary entity that is running the experiment with $\mathcal{A}$.) Then, a ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$.*
3. *$\mathcal{A}$ outputs a bit b'.*
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. We write $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1$ if the output is 1 and in this case we say that $\mathcal{A}$ $\mathsf{succeeded}$.*

One should think of $\mathcal{A}$ as trying to guess the value of b that is chosen in the experiment, and $\mathcal{A}$ succeeds when its guess b' is correct. Observe that it is always possible for $\mathcal{A}$ to succeed in the experiment with probability one half by just guessing b' randomly. The question is whether it is possible for $\mathcal{A}$ to do any better than this. The alternate definition we now give states that an encryption scheme is perfectly secret if *no* adversary $\mathcal{A}$ can succeed with probability any better than one half. We stress that, as is the case throughout this chapter, there is no limitation whatsoever on the computational power of $\mathcal{A}$.

DEFINITION 2.4 *An encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over a message space $\mathcal{M}$ is* perfectly secret *if for every adversary $\mathcal{A}$ it holds that*

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1\right] = \frac{1}{2}.$$

The following proposition states that Definition 2.4 is equivalent to Definition 2.1. We leave the proof of the proposition as an exercise.

PROPOSITION 2.5 *Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme over a message space $\mathcal{M}$. Then $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is perfectly secret with respect to Definition 2.1 if and only if it is perfectly secret with respect to Definition 2.4.*

2.2 The One-Time Pad (Vernam's Cipher)

In 1917, Vernam patented a cipher now called the *one-time pad* that obtains perfect secrecy. There was no proof of this fact at the time (in fact, there was not yet a notion of what perfect secrecy was). Rather, approximately 25 years later, Shannon introduced the notion of perfect secrecy and demonstrated that the one-time pad achieves this level of security.

Let $a \oplus b$ denote the *bitwise exclusive-or* (XOR) of two binary strings a and b (i.e., if $a = a_1, \ldots, a_\ell$ and $b = b_1, \ldots, b_\ell$, then $a \oplus b = a_1 \oplus b_1, \ldots, a_\ell \oplus b_\ell$). The one-time pad encryption scheme is defined as follows:

1. Fix an integer $\ell > 0$. Then the message space $\mathcal{M}$, key space $\mathcal{K}$, and ciphertext space $\mathcal{C}$ are all equal to $\{0,1\}^\ell$ (i.e., the set of all binary strings of length ℓ).

2. The key-generation algorithm Gen works by choosing a string from $\mathcal{K} = \{0,1\}^\ell$ according to the uniform distribution (i.e., each of the 2^ℓ strings in the space is chosen as the key with probability exactly $2^{-\ell}$).

3. Encryption Enc works as follows: given a key $k \in \{0,1\}^\ell$ and a message $m \in \{0,1\}^\ell$, output $c := k \oplus m$.

4. Decryption Dec works as follows: given a key $k \in \{0,1\}^\ell$ and a ciphertext $c \in \{0,1\}^\ell$, output $m := k \oplus c$.

Before discussing the security of the one-time pad, we note that for every k and every m it holds that $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = k \oplus k \oplus m = m$ and so the one-time pad constitutes a legal encryption scheme.

Intuitively, the one-time pad is perfectly secret because given a ciphertext c, there is no way an adversary can know which plaintext m it originated from. In order to see why this is true, notice that for every possible m there exists a key k such that $c = \mathsf{Enc}_k(m)$; namely, take $k = m \oplus c$. Furthermore, each key is chosen with uniform probability (and hidden from the adversary) and so no key is more likely than any other. Combining the above, we obtain that c reveals nothing whatsoever about which plaintext m was encrypted, because every plaintext is equally likely to have been encrypted. We now prove this intuition formally:

THEOREM 2.6 *The one-time pad encryption scheme is perfectly-secret.*

PROOF Fix some distribution over $\mathcal{M}$ and fix an arbitrary $m \in \mathcal{M}$ and $c \in \mathcal{C}$. The key observation is that for the one-time pad,

$$\begin{aligned}\Pr[C = c \mid M = m] &= \Pr[M \oplus K = c \mid M = m] \\ &= \Pr[m \oplus K = c] = \Pr[K = m \oplus c] = \frac{1}{2^\ell}.\end{aligned}$$

Since this holds for all distributions and all m, we have that for every probability distribution over $\mathcal{M}$, every $m_0, m_1 \in \mathcal{M}$ and every $c \in \mathcal{C}$,

$$\Pr[C = c \mid M = m_0] = \frac{1}{2^\ell} = \Pr[C = c \mid M = m_1].$$

By Lemma 2.3, this implies that the encryption scheme is perfectly secret. ■

We conclude that perfect secrecy is attainable. Unfortunately, the one-time pad encryption scheme has a number of drawbacks. Most prominent is that *the key is required to be as long as the message.* First and foremost, this means that a long key must be securely stored, something that is highly problematic in practice and often not achievable. In addition, this limits applicability of the scheme if we want to send very long messages (as it may be difficult to securely store a very long key) or if we don't know in advance an upper bound on how long the message will be (since we can't share a key of unbounded length). Moreover, the one-time pad scheme — as the name indicates — *is only "secure" if used once (with the same key).* Although we did not yet define a notion of security when multiple messages are encrypted, it is easy to see informally that encrypting more than one message leaks a lot of information. In particular, say two messages m, m' are encrypted using the same key k. An adversary who obtains $c = m \oplus k$ and $c' = m' \oplus k$ can compute

$$\begin{aligned} c \oplus c' &= (m \oplus k) \oplus (m' \oplus k) \\ &= m \oplus m' \end{aligned}$$

and thus learn something about the exclusive-or of the two messages. While this may not seem very significant, it is enough to rule out any claims of perfect secrecy when encrypting two messages. Furthermore, if the messages correspond to English-language text, then given the exclusive-or of two sufficiently-long messages, it has been shown to be possible to perform frequency analysis (as in the previous chapter, though more complex) and recover the messages themselves.

2.3 Limitations of Perfect Secrecy

In this section, we show that the aforementioned limitations of the one-time pad encryption scheme are *inherent.* Specifically, we prove that *any* perfectly-secret encryption scheme must have a key space that is at least as large as the message space. If the key space consists of fixed-length keys, and the message space consists of all messages of some fixed length, this implies that the key must be at least as long as the message. Thus, the problem of a large key length is not specific to the one-time pad, but is inherent to any scheme achieving perfect secrecy. (The other limitation regarding the fact that the key can only be used once is also inherent in the context of perfect secrecy; see, e.g., Exercise 2.9.)

THEOREM 2.7 *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a perfectly-secret encryption scheme over a message space* $\mathcal{M}$*, and let* $\mathcal{K}$ *be the key space as determined by* Gen*. Then* $|\mathcal{K}| \geq |\mathcal{M}|$.

PROOF We show that if $|\mathcal{K}| < |\mathcal{M}|$ then the scheme is not perfectly secret. Assume $|\mathcal{K}| < |\mathcal{M}|$. Consider the uniform distribution over $\mathcal{M}$ and let $c \in \mathcal{C}$ be a ciphertext that occurs with non-zero probability. Let $\mathcal{M}(c)$ be the set of all possible messages which are possible decryptions of c; that is

$$\mathcal{M}(c) \stackrel{\text{def}}{=} \{\hat{m} \mid \hat{m} = \mathsf{Dec}_{\hat{k}}(c) \text{ for some } \hat{k} \in \mathcal{K}\}.$$

Clearly $|\mathcal{M}(c)| \leq |\mathcal{K}|$ since for each message $\hat{m} \in \mathcal{M}(c)$ we can identify at least one key $\hat{k} \in \mathcal{K}$ for which $\hat{m} = \mathsf{Dec}_{\hat{k}}(c)$. (Recall that we assume Dec is deterministic.) Under the assumption that $|\mathcal{K}| < |\mathcal{M}|$, this means that there is some $m' \in \mathcal{M}$ such that $m' \notin \mathcal{M}(c)$. But then

$$\Pr[M = m' \mid C = c] = 0 \neq \Pr[M = m'],$$

and so the scheme is not perfectly secret. ■

Perfect secrecy at a lower price? The above theorem shows an inherent limitation of schemes that achieve perfect secrecy. Even so, it is often claimed by individuals and/or companies that they have developed a radically new encryption scheme that is unbreakable and achieves the security level of the one-time pad without using long keys. The above proof demonstrates that such claims *cannot* be true; the person claiming them either knows very little about cryptography or is blatantly lying.

2.4 * Shannon's Theorem

In his breakthrough work on perfect secrecy, Shannon also provided a characterization of perfectly-secret encryption schemes. As we shall see below, this characterization says that, assuming $|\mathcal{K}| = |\mathcal{M}| = |\mathcal{C}|$, the key-generation algorithm Gen must choose a secret key *uniformly* from the set of all possible keys (as in the one-time pad), and that for every plaintext message and ciphertext there exists a *single* key mapping the plaintext to the ciphertext (again, as in the one-time pad). Beyond being interesting in its own right, this theorem is a powerful tool for proving (or contradicting) the perfect secrecy of suggested schemes. We discuss this further after the proof.

As before, we assume that the probability distributions over $\mathcal{M}$ and $\mathcal{C}$ are such that all $m \in \mathcal{M}$ and $c \in \mathcal{C}$ are assigned non-zero probabilities. The theorem here considers the special case when $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$, meaning that the sets of plaintexts, keys, and ciphertexts are all of the same size. We have already seen that $|\mathcal{K}| \geq |\mathcal{M}|$. It is easy to see that $|\mathcal{C}|$ must also be at least the size of $|\mathcal{M}|$ because otherwise for every key, there must be two plaintexts that

are mapped to a single ciphertext (making it impossible to unambiguously decrypt). Therefore, in some sense, the case of $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$ is the "most efficient". We are now ready to state the theorem:

THEOREM 2.8 (Shannon's theorem) *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be an encryption scheme over a message space* $\mathcal{M}$ *for which* $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$. *The scheme is perfectly secret if and only if:*

1. *Every key* $k \in \mathcal{K}$ *is chosen with equal probability* $1/|\mathcal{K}|$ *by algorithm* Gen.
2. *For every* $m \in \mathcal{M}$ *and every* $c \in \mathcal{C}$, *there exists a unique key* $k \in \mathcal{K}$ *such that* $\mathsf{Enc}_k(m)$ *outputs* c.

PROOF The intuition behind the proof of this theorem is as follows. First, if a scheme fulfills item (2) then a given ciphertext c could be the result of encrypting any possible plaintext m (this holds because for every m there exists a key k mapping it to c). Combining this with the fact that exactly one key maps each m to c, and by item (1) each key is chosen with the same probability, perfect secrecy can be shown as in the case of the one-time pad. For the other direction, the intuition is that if $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$ then there must be exactly one key mapping each m to each c. (Otherwise, either some m is not mapped to a given c contradicting perfect secrecy, or some m is mapped by more than one key to c, resulting in another m' not being mapped to c, again contradicting perfect secrecy.) Given this, it must hold that each key is chosen with equal probability or some plaintexts would be more likely than others, contradicting perfect secrecy. The formal proof follows.

Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be as in the theorem. For simplicity, we assume Enc is deterministic. We first prove that if $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is perfectly secret, then items (1) and (2) hold. As in the proof of Theorem 2.7, it is not hard to see that for every $m \in \mathcal{M}$ and $c \in \mathcal{C}$, there exists *at least one* key $k \in \mathcal{K}$ such that $\mathsf{Enc}_k(m) = c$. (Otherwise, $\Pr[M = m \mid C = c] = 0 \neq \Pr[M = m]$.) For a fixed m, consider now the set $\{\mathsf{Enc}_k(m)\}_{k \in \mathcal{K}}$. By what we have just said, $|\{\mathsf{Enc}_k(m)\}_{k \in \mathcal{K}}| \geq |\mathcal{C}|$ (because for every $c \in \mathcal{C}$ there exists a $k \in \mathcal{K}$ such that $\mathsf{Enc}_k(m) = c$). In addition, we trivially have $|\{\mathsf{Enc}_k(m)\}_{k \in \mathcal{K}}| \leq |\mathcal{C}|$. We conclude that

$$|\{\mathsf{Enc}_k(m)\}_{k \in \mathcal{K}}| = |\mathcal{C}|.$$

Since $|\mathcal{K}| = |\mathcal{C}|$, it follows that $|\{\mathsf{Enc}_k(m)\}_{k \in \mathcal{K}}| = |\mathcal{K}|$. This implies that there are no distinct keys $k_1, k_2 \in \mathcal{K}$ with $\mathsf{Enc}_{k_1}(m) = \mathsf{Enc}_{k_2}(m)$. Since m was arbitrary, we see that for every m and c, there exists *at most* one key $k \in \mathcal{K}$ such that $\mathsf{Enc}_k(m) = c$. Combining the above (i.e., the existence of at least one key and at most one key), we obtain item (2).

We proceed to show that for every $k \in \mathcal{K}$, $\Pr[K = k] = 1/|\mathcal{K}|$. Let $n = |\mathcal{K}|$ and $\mathcal{M} = \{m_1, \ldots, m_n\}$ (recall, $|\mathcal{M}| = |\mathcal{K}| = n$), and fix a ciphertext c. Then, we can label the keys $k_1, \ldots, k_n$ so that for every i $(1 \leq i \leq n)$ it holds

that $\mathsf{Enc}_{k_i}(m_i) = c$. This labeling can be carried out because, as just shown, for every c and m_i there exists a *unique* key k_i such that $\mathsf{Enc}_{k_i}(m_i) = c$, and furthermore these keys are distinct for distinct m_i, m_j (since otherwise unambiguous decryption would be impossible). By perfect secrecy we have that for every i:

$$\begin{aligned}\Pr[M = m_i] &= \Pr[M = m_i \mid C = c] \\ &= \frac{\Pr[C = c \mid M = m_i] \cdot \Pr[M = m_i]}{\Pr[C = c]} \\ &= \frac{\Pr[K = k_i] \cdot \Pr[M = m_i]}{\Pr[C = c]},\end{aligned}$$

where the second equality is by Bayes' theorem and the third equality holds by the labeling above (i.e., k_i is the unique key that maps m_i to c). From the above, it follows that for every i,

$$\Pr[K = k_i] = \Pr[C = c].$$

Therefore, for every i and j, $\Pr[K = k_i] = \Pr[C = c] = \Pr[K = k_j]$ and so all keys are chosen with the same probability. We conclude that keys are chosen according to the uniform distribution. That is, for every k, $\Pr[K = k_i] = 1/|\mathcal{K}|$ as required.

We now prove the other direction of the theorem. Assume that every key is obtained with probability $1/|\mathcal{K}|$ and that for every $m \in \mathcal{M}$ and $c \in \mathcal{C}$ there is a unique key $k \in \mathcal{K}$ such that $\mathsf{Enc}_k(m) = c$. This immediately implies that for every m and c,

$$\Pr[C = c \mid M = m] = \frac{1}{|\mathcal{K}|}$$

irrespective of the probability distribution over $\mathcal{M}$. Thus, for every probability distribution over $\mathcal{M}$, every $m, m' \in \mathcal{M}$, and every $c \in \mathcal{C}$ we have

$$\Pr[C = c \mid M = m] = \frac{1}{|\mathcal{K}|} = \Pr[C = c \mid M = m'],$$

and so by Lemma 2.3 the encryption scheme is perfectly secret. ■

Uses of Shannon's theorem. Theorem 2.8 is of interest in its own right in that it essentially gives a complete characterization of perfectly-secret encryption schemes. In addition, since items (1) and (2) have nothing to do with the probability distribution over the set of plaintexts $\mathcal{M}$, the theorem implies that if there exists an encryption scheme that provides perfect secrecy for a specific probability distribution over $\mathcal{M}$ then it actually provides perfect secrecy in general (i.e., for all probability distributions over $\mathcal{M}$). Finally, Shannon's theorem is extremely useful for proving whether a given scheme is or is not perfectly secret. Item (1) is easy to confirm and item (2) can be demonstrated

(or contradicted) without analyzing any probabilities (in contrast to working with, say, Definition 2.1). For example, the perfect secrecy of the one-time pad (Theorem 2.6) is trivial to prove using Shannon's theorem. We warn, however, that Theorem 2.8 only holds if $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$, and so one must be careful to apply it only in this case.

2.5 Summary

This completes our treatment of perfectly-secret encryption. The main lesson of this chapter is that *perfect secrecy is attainable*, meaning that there exist encryption schemes with the property that the ciphertext reveals absolutely nothing about the plaintext, even to an adversary with unlimited computational power. However, all such schemes have the limitation that the key must be at least as long as the message. In practice, therefore, perfectly-secret encryption is rarely used. We remark that it is rumored that the "red phone" linking the White House and the Kremlin during the Cold War was protected using one-time pad encryption. Of course, the governments of the US and USSR could exchange extremely long random keys without great difficulty, and therefore practically use the one-time pad. However, in most settings (especially commercial ones), the limitation regarding the key length makes the one-time pad or any other perfectly-secret scheme unusable.

References and Additional Reading

The notion of perfectly-secret encryption was introduced and studied in the ground-breaking work of Shannon [127]. In addition to introducing the notion, he proved that the one-time pad (originally introduced by Vernam [141]) is perfectly secret, and also proved the theorems characterizing perfectly-secret schemes (and their implied limitations). Stinson [138] contains further discussion of perfect secrecy.

In this chapter we have briefly studied perfectly-secure *encryption*. There are other cryptographic problems that can also be solved with "perfect" security. A notable example is the problem of message authentication where the aim is to prevent an adversary from modifying a message (in an undetectable manner) en route from one party to another; we study this problem in depth in Chapter 4 in the computational setting. The reader interested in learning about perfectly-secure message authentication is referred to the paper by Stinson [136], the survey by Simmons [134], or the first edition of Stinson's textbook [137, Chapter 10] for further information.

Exercises

2.1 Prove the second direction of Lemma 2.2.

2.2 Prove or refute: For every encryption scheme that is perfectly secret it holds that for every distribution over the message space $\mathcal{M}$, every $m, m' \in \mathcal{M}$, and every $c \in \mathcal{C}$:

$$\Pr[M = m \mid C = c] = \Pr[M = m' \mid C = c].$$

2.3 When using the one-time pad (Vernam's cipher) with the key $k = 0^\ell$, it follows that $\mathsf{Enc}_k(m) = k \oplus m = m$ and the message is effectively sent in the clear! It has therefore been suggested to improve the one-time pad by only encrypting with a key $k \neq 0^\ell$ (i.e., to have Gen choose k uniformly at random from the set of *non-zero* keys of length ℓ). Is this an improvement? In particular, is it still perfectly secret? Prove your answer. If your answer is positive, explain why the one-time pad is not described in this way. If your answer is negative, reconcile this with the fact that encrypting with 0^ℓ doesn't change the plaintext.

2.4 In this exercise, we study conditions under which the shift, mono-alphabetic substitution, and Vigenére ciphers are perfectly secret:

(a) Prove that if only a single character is encrypted, then the shift cipher is perfectly secret.

(b) What is the largest plaintext space $\mathcal{M}$ you can find for which the mono-alphabetic substitution cipher provides perfect secrecy? (Note: $\mathcal{M}$ need not contain only valid English words.)

(c) Show how to use the Vigenére cipher to encrypt any word of length t so that perfect secrecy is obtained (note: you can choose the length of the key). Prove your answer.

Reconcile this with the attacks that were shown in the previous chapter.

2.5 Prove or refute: Every encryption scheme for which the size of the key space equals the size of the message space, and for which the key is chosen uniformly from the key space, is perfectly secret.

2.6 Say encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ satisfies Definition 2.1 for all distributions over $\mathcal{M}$ *that assign non-zero probability to each* $m \in \mathcal{M}$ (as per the simplifying convention used in this chapter). Show that the scheme satisfies the definition for *all* distributions over $\mathcal{M}$ (i.e., including those that assign zero probability to some messages in $\mathcal{M}$). Conclude that the scheme is also perfectly secret for any message space $\mathcal{M}' \subset \mathcal{M}$.

2.7 Prove that Definition 2.1 implies Definition 2.4.

Hint: Use Exercise 2.6 to argue that perfect secrecy holds for the uniform distribution over any two plaintexts (and in particular, the two messages output by $\mathcal{A}$ in the experiment). Then apply Lemma 2.3.

2.8 Prove the second direction of Proposition 2.5. That is, prove that Definition 2.4 implies Definition 2.1.

Hint: If a scheme Π is not perfectly secret with respect to Definition 2.1, then Lemma 2.3 shows that there exist messages $m_0, m_1 \in \mathcal{M}$ and $c \in \mathcal{C}$ for which $\Pr[C = c \mid M = m_0] \neq \Pr[C = c \mid M = m_1]$. Use these m_0 and m_1 to construct an $\mathcal{A}$ for which $\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1] > \frac{1}{2}$.

2.9 Consider the following definition of perfect secrecy for the encryption of *two* messages. An encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over a message space $\mathcal{M}$ is *perfectly-secret for two messages* if for all distributions over $\mathcal{M}$, all $m, m' \in \mathcal{M}$, and all $c, c' \in \mathcal{C}$ with $\Pr[C = c \wedge C' = c'] > 0$:

$$\Pr\left[M = m \wedge M' = m' \mid C = c \wedge C' = c'\right] = \Pr[M = m \wedge M' = m'],$$

where m and m' are sampled independently from the same distribution over $\mathcal{M}$. Prove that *no* encryption scheme satisfies this definition.

Hint: Take $m \neq m'$ but $c = c'$.

2.10 Consider the following definition of perfect secrecy for the encryption of two messages. Encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over a message space $\mathcal{M}$ is *perfectly-secret for two messages* if for all distributions over $\mathcal{M}$, all $m, m' \in \mathcal{M}$ with $m \neq m'$, and all $c, c' \in \mathcal{C}$ with $c \neq c'$ and $\Pr[C = c \wedge C' = c'] > 0$:

$$\begin{aligned}\Pr[M = m \wedge M' = m' \mid C = c \wedge C' = c'] \\ = \Pr[M = m \wedge M' = m' \mid M \neq M'],\end{aligned}$$

where m and m' are sampled independently from the same distribution over $\mathcal{M}$. Show an encryption scheme that provably satisfies this definition. How long are the keys compared to the length of a message?

Hint: The encryption scheme you propose need not be "efficient".

2.11 Prove that we may assume that the key-generation algorithm Gen always chooses a key uniformly from the key space $\mathcal{K}$.

Hint: Let $\mathcal{K}$ denote the set of all possible random tapes for the randomized algorithm Gen.

2.12 Assume that we require only that an encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over a message space $\mathcal{M}$ satisfies the following: for all $m \in \mathcal{M}$, the probability that $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m$ is at least 2^{-t}. (This probability is taken over choice of k as well as any randomness that may be used

during encryption or decryption.) Show that perfect secrecy (as in Definition 2.1) can be achieved with $|\mathcal{K}| < |\mathcal{M}|$ when $t \geq 1$. Can you prove a lower bound on the required size of $\mathcal{K}$?

2.13 Prove an analogue of Theorem 2.7 for the case of "almost perfect" secrecy. That is, let $\varepsilon < 1$ be a constant and say we only require that for any distribution over $\mathcal{M}$, any $m \in \mathcal{M}$, and any $c \in \mathcal{C}$;

$$|\Pr[M = m \mid C = c] - \Pr[M = m]| < \varepsilon.$$

Prove a lower bound on the size of the key space $\mathcal{K}$ relative to $\mathcal{M}$ for any encryption scheme that meets this definition.

Hint: Consider the uniform distribution over $\mathcal{M}$ and fix a ciphertext c. Then show that for a $(1-\varepsilon)$ fraction of the messages $m \in \mathcal{M}$, there must exist a key mapping m to c.

Part II

Private-Key (Symmetric) Cryptography

Chapter 3

Private-Key Encryption and Pseudorandomness

In this chapter, we will study the notion of *pseudorandomness* — the idea that things can "look" completely random (in a sense we precisely define) even though they are not — and see how this can be used to achieve secure encryption beating the bounds of the previous chapter. Specifically, we will see encryption schemes whereby a short key (say, some hundreds of bits long) can be used to securely encrypt many long messages (say, gigabytes in total). Such schemes are able to bypass the inherent limitations of perfect secrecy because they achieve the weaker but sufficient notion of *computational* secrecy. Before commencing our discussion of private-key encryption, we first examine the computational approach to cryptography in general. The computational approach will be used in the rest of the book, and is the basis of modern cryptography.

3.1 A Computational Approach to Cryptography

In the previous two chapters we have studied what can be called *classical cryptography*. We began with a brief look at some historical ciphers, with a focus on how they can be broken and what can be learned from these attacks. In Chapter 2, we then proceeded to present cryptographic schemes that can be mathematically proven secure (with respect to some particular definition of security), even when the adversary has unlimited computational power. Such schemes are called *information-theoretically secure*, or *perfectly secure*, because their security is due to the fact that the adversary simply does not have enough "information" to succeed in its attack, regardless of the adversary's computational power.[1] In particular, as we have discussed, the ciphertext in a perfectly-secret encryption scheme does not contain any information about the plaintext (assuming the key is unknown).

[1] The term "information" has a rigorous, mathematical meaning, but we use it here in an informal manner.

Information-theoretic security stands in stark contrast to *computational security* that is the aim of most modern cryptographic constructions. Restricting ourselves to the case of private-key encryption (though everything we say applies more generally), modern encryption schemes have the property that they *can* be broken given enough time and computation, and so they do not satisfy Definition 2.1. Nevertheless, under certain assumptions, the amount of computation needed to break these encryption schemes would take more than many lifetimes to carry out even using the fastest available supercomputers. For all practical purposes, this level of security suffices.

Computational security is weaker than information-theoretic security. It also currently relies on unproven assumptions, whereas no assumptions are needed to achieve the latter.[2] Even granting the fact that computational security suffices for all practical purposes, why do we give up on the idea of achieving perfect security? The results of Section 2.3 give one reason why modern cryptography has taken this route. In that section, we showed that perfectly-secret encryption schemes suffer from severe lower bounds on the key length; namely, the key must be as long as the combined length of *all* messages ever encrypted using this key. Similar negative results hold for other cryptographic tasks when information-theoretic security is required. Thus, despite its mathematical appeal, it is necessary to compromise on perfect security in order to obtain practical cryptographic schemes. We stress that although we cannot obtain perfect security, this does not mean that we do away with the rigorous mathematical approach. Rather, definitions and proofs are still essential, and the only difference is that now we consider *weaker*, but still meaningful, definitions of security.

3.1.1 The Basic Idea of Computational Security

Kerckhoffs is best known for his principle that cryptographic designs should be made public. However, he actually spelled out six principles, the following of which is very relevant to our discussion here:

> *A [cipher] must be practically, if not mathematically, indecipherable.*

Although he could not have stated it in this way at the time, this principle of Kerckhoffs essentially says that it is not necessary to use a perfectly-secret encryption scheme, but it instead suffices to use a scheme that cannot be broken in "reasonable time" with any "reasonable probability of success" (in Kerckhoffs' language, a scheme that is "practically indecipherable"). In more concrete terms, it suffices to use an encryption scheme that can (in theory) be

[2] In theory, it is possible that these assumptions might one day be removed. Unfortunately, however, our current state of knowledge requires us to make assumptions in order to prove computational security of any cryptographic construction. For those familiar with the $\mathcal{P}$ versus $\mathcal{NP}$ question it is worth noting that any unconditional proof of a computationally secure cryptographic construction would require, in particular, proving that $\mathcal{P} \neq \mathcal{NP}$.

broken, but that *cannot* be broken with probability better than 10^{-30} in 200 years using the fastest available supercomputer. In this section we present a framework for making formal statements about cryptographic schemes that are "practically unbreakable".

The computational approach incorporates two relaxations of the notion of perfect security:

1. *Security is only preserved against* efficient *adversaries that run in a feasible amount of time,* and
2. *Adversaries can potentially succeed with some* very small probability *(that is small enough so that we are not concerned that it will ever really happen).*

To obtain a meaningful theory, we need to precisely define what is meant by the above. There are two common approaches for doing so: the *concrete approach* and the *asymptotic approach.* We explain these now.

The concrete approach. The concrete approach quantifies the security of a given cryptographic scheme by explicitly bounding the maximum success probability of any adversary running for at most some specified amount of time. That is, let t, ε be positive constants with $\varepsilon \leq 1$. Then a concrete definition of security would, roughly speaking, take the following form:

> *A scheme is* (t, ε)-secure *if every adversary running for time at most* t *succeeds in breaking the scheme with probability at most* ε*.*

(Of course, the above serves only as a general template, and for the above statement to make sense we need to define exactly what it means to "break" the scheme.) As an example, one might want to use a scheme with the guarantee that no adversary running for at most 200 years using the fastest available supercomputer can succeed in breaking the scheme with probability better than 10^{-30}. Or, it may be more convenient to measure running time in terms of CPU cycles, and to use a scheme such that no adversary running for at most 2^{80} cycles can break the scheme with probability better than 2^{-64}.

It is instructive to get a feel for values of t, ε that are typical of modern cryptographic schemes.

Example 3.1

Modern private-key encryption schemes are generally assumed to give almost optimal security in the following sense: when the key has length n, an adversary running in time t (measured in, say, computer cycles) can succeed in breaking the scheme with probability at most $t/2^n$. (We will see later why this is indeed optimal.) Computation on the order of $t = 2^{60}$ is barely within reach today. Running on a 1GHz computer (that executes 10^9 cycles per second), 2^{60} CPU cycles require $2^{60}/10^9$ seconds, or about 35 years. Using many supercomputers in parallel may bring this down to a few years.

A typical value for the key length, however, might be $n = 128$. The difference between 2^{60} and 2^{128} is a *multiplicative factor* of 2^{68} which is a number containing about 21 decimal digits. To get a feeling for how big this is, note that according to physicists' estimates the number of seconds since the big bang is in the order of 2^{58}.

An event that occurs once every hundred years can be roughly estimated to occur with probability 2^{-30} in any given second. Something that occurs with probability 2^{-60} in any given second is 2^{30} times *less* likely, and might be expected to occur roughly once every 100 billion years. Thus a prudent choice of parameters would be $t = 2^{80}$ and $\varepsilon = 2^{-48}$ (implying that the key must be at least 128 bits long). ♢

The concrete approach can be useful in practice, since concrete guarantees of the above type are typically what users of a cryptographic scheme are ultimately interested in. However, one must be careful in interpreting concrete security guarantees. As one example, if it is claimed that no adversary running for 5 years can break a given scheme with probability better than ε, we still must ask: what type of computing power (e.g., desktop PC, supercomputer, network of hundreds of computers) does this assume? Does this take into account future advances in computing power (which, by Moore's Law, roughly doubles every 18 months)? Does this assume "off-the-shelf" algorithms will be used or dedicated software optimized for the attack? Furthermore, such a guarantee says little about the success probability of an adversary running for 2 years (other than the fact that it can be at most ε) and says nothing about the success probability of an adversary running for 10 years.

When using the concrete security approach, schemes can be (t, ε)-secure but never just *secure*. More to the point, for what ranges of t, ε should we say that a (t, ε)-secure scheme is "secure"? There is no clear answer to this, as a security guarantee that may suffice for the average user may not suffice when encrypting classified government documents.

The asymptotic approach. The asymptotic approach is the one we will take in this book. This approach, rooted in complexity theory, views the running time of the adversary as well as its success probability as *functions* of some parameter rather than as concrete numbers. Specifically, a cryptographic scheme will incorporate a security parameter which is an integer n. When honest parties initialize the scheme (i.e., when they generate keys), they choose some value n for the security parameter; this value is assumed to be known to any adversary attacking the scheme. The running time of the adversary (and the running time of the honest parties) as well as the adversary's success probability are all viewed as functions of n. Then:

1. We equate the notion of "feasible strategies" or "efficient algorithms" with probabilistic algorithms running in time *polynomial in* n. (We sometimes use PPT to stand for *probabilistic polynomial-time*.) This means that for some constants a, c the algorithm runs in time $a \cdot n^c$ on

security parameter n. We require that honest parties run in polynomial time, and will only be concerned with achieving security against polynomial-time adversaries. We stress that the adversary, though required to run in polynomial time, may be much more powerful (and run much longer) than the honest parties. Furthermore, adversarial strategies that require a super-polynomial amount of time are not considered realistic threats (and so are essentially ignored).

2. We equate the notion of "small probability of success" with success probabilities *smaller than any inverse polynomial in* n, meaning that for every constant c the adversary's success probability is smaller than n^{-c} for large enough values of n (see Definition 3.4). A function that grows slower than any inverse polynomial is called negligible.

A definition of asymptotic security thus takes the following general form:

> *A scheme is* secure *if every* PPT *adversary succeeds in breaking the scheme with only negligible probability.*

Although very clean from a theoretical point of view (since we can actually speak of a scheme being secure or not), it is important to understand that the asymptotic approach only "guarantees security" for large enough values of n, as the following example should make clear.

Example 3.2
Say we have a scheme that is secure. Then it may be the case that an adversary running for n^3 minutes can succeed in "breaking the scheme" with probability $2^{40} \cdot 2^{-n}$ (which is a negligible function of n). When $n \leq 40$ this means that an adversary running for 40^3 minutes (about 6 weeks) can break the scheme with probability 1, so such values of n are not going to be very useful. Even for $n = 50$ an adversary running for 50^3 minutes (about 3 months) can break the scheme with probability roughly 1/1000, which may not be acceptable. On the other hand, when $n = 500$ an adversary running for more than 200 years breaks the scheme only with probability roughly 2^{-500}. ♢

As indicated by the previous example, we can view a larger security parameter as providing a "greater" level of security. For the most part, the security parameter determines the length of the key used by the honest parties, and we thus have the familiar concept that the longer the key, the higher the security. The ability to "increase security" by taking a larger value for the security parameter has important practical ramifications, since it enables honest parties to defend against increases in computing power as well as algorithmic advances. The following example gives a sense of how this might play out in practice.

Example 3.3
Let us see the effect that the availability of faster computers might have on security in practice. Say we have a cryptographic scheme where honest parties are required to run for $10^6 \cdot n^2$ cycles, and for which an adversary running for $10^8 \cdot n^4$ cycles can succeed in "breaking" the scheme with probability $2^{20} \cdot 2^{-n}$. (The numbers in this example are designed to make calculations easier, and are not meant to correspond to any existing cryptographic scheme.)

Say all parties are using a 1Ghz computer and $n = 50$. Then honest parties run for $10^6 \cdot 2500$ cycles, or 2.5 seconds, and an adversary running for $10^8 \cdot (50)^4$ cycles, or roughly 1 week, can break the scheme with probability only 2^{-30}.

Now say a 16Ghz computer becomes available, and all parties upgrade. Honest parties can increase n to 100 (which requires generating a fresh key) and still improve their running time to 0.625 seconds (i.e., $10^6 \cdot 100^2$ cycles at $16 \cdot 10^9$ cycles/second). In contrast, the adversary now has to run for 10^7 seconds, or more than *16 weeks*, to achieve success probability 2^{-80}. The effect of a faster computer has been to make the adversary's job *harder*. $\diamondsuit$

The asymptotic approach has the advantage of not depending on any specific assumptions regarding, e.g., the type of computer an adversary uses (this is a consequence of the *Church-Turing thesis* from complexity theory, which basically states that the relative speeds of all sufficiently-powerful computing devices are polynomially related). On the other hand, as the above examples demonstrate, it is necessary in practice to understand exactly what level of concrete security is implied by a particular asymptotically-secure scheme. This is because the honest parties must pick a concrete value of n to use, and so cannot rely on assurances of what happens "for large enough values of n". The task of determining the value of the security parameter to use is complex and depends on the scheme in question as well as other considerations. Fortunately, it is usually relatively easy to translate a guarantee of asymptotic security into a concrete security guarantee.

From here on, we use the asymptotic approach only. Nevertheless, as the above example shows, all the results in this book can be cast as concrete security results as well.

A technical remark. As we have mentioned, we view the running time of the adversary and the honest parties as a function of n. To be consistent with the standard convention in algorithms and complexity theory, where the running time of an algorithm is measured as a function of the length of its input, we will provide the adversary and the honest parties with the security parameter in *unary* as 1^n (i.e., a string of n 1's) when necessary.

Necessity of the Relaxations

As we have seen, computational security introduces two relaxations of the notion of perfect security: first, security is guaranteed only against efficient (i.e., polynomial-time) adversaries; second, a small (i.e., negligible) probability of success is allowed. Both of these relaxations are essential for achieving practical cryptographic schemes, and in particular for bypassing the negative results for perfectly-secret encryption. We will now informally discuss why this is the case. Assume we have an encryption scheme where the size of the key space $\mathcal{K}$ is much smaller than the size of the message space $\mathcal{M}$ (which, as we saw in the previous chapter, means that the scheme cannot be perfectly secret). Two attacks, lying at opposite extremes, apply regardless of how the encryption scheme is constructed:

- Given a ciphertext c, an adversary can decrypt c using all keys $k \in \mathcal{K}$. This gives a list of all possible messages to which c can possibly correspond. Since this list cannot contain all of $\mathcal{M}$ (because $|\mathcal{K}| < |\mathcal{M}|$), this leaks *some* information about the message that was encrypted.

  Moreover, say the adversary carries out a known-plaintext attack and learns that ciphertexts $c_1, \ldots, c_\ell$ correspond to the messages $m_1, \ldots, m_\ell$, respectively. The adversary can again try decrypting each of these ciphertexts with all possible keys until it finds a key k for which $\mathsf{Dec}_k(c_i) = m_i$ for all i. This key will be unique with high probability, in which case the adversary has found the key that the honest parties are using.[3] Subsequent usage of this key will therefore be insecure.

  The type of attack is known as *brute-force search* and allows the adversary to succeed with probability essentially 1 in time linear in $|\mathcal{K}|$.

- Consider again the case where the adversary learns that ciphertexts $c_1, \ldots, c_\ell$ correspond to the messages $m_1, \ldots, m_\ell$. The adversary can *guess* a key $k \in \mathcal{K}$ at random and check to see whether $\mathsf{Dec}_k(c_i) = m_i$ for all i. If so, we again expect that with high probability k is the key that the honest parties are using.

  Here the adversary runs in essentially constant time and succeeds with non-zero (although very small) probability of roughly $1/|\mathcal{K}|$.

It follows that if we wish to encrypt many messages using a single short key, security can only be achieved if we limit the running time of the adversary (so that the adversary does not have time to carry out a brute-force search) and also allow a very small probability of success without considering it a break (so that the second "attack" is ruled out). Thus, both of the aforementioned

[3] Technically speaking, the key may not actually be unique (for example, the same key may be represented by more than one string). Nevertheless, if this is the case then an equivalent key will be found, yielding the same effect.

relaxations are essential. We remark that an additional consequence of the above discussion is that the keyspace in any secure encryption scheme must be large enough so that the adversary cannot traverse it. Stated more formally, the size of the keyspace must be super-polynomial in the security parameter.

3.1.2 Efficient Algorithms and Negligible Success

In the previous section we have outlined the approach of asymptotic security that we will be taking in this book. Students who have not had significant prior exposure to algorithms or complexity theory may not be comfortable with the notions of "polynomial-time algorithms", "probabilistic (or randomized) algorithms", or "negligible probabilities", and often find the asymptotic approach confusing. In this section we revisit the asymptotic approach in more detail, and slightly more formally. Students who are already comfortable with what was described in the previous section are welcome to skip ahead to Section 3.1.3 and refer back here as needed.

Efficient Computation

We have defined efficient computation as that which can be carried out in *probabilistic polynomial time* (sometimes abbreviated PPT). An algorithm A is said to run in polynomial time if there exists a polynomial $p(\cdot)$ such that, for every input $x \in \{0,1\}^*$, the computation of $A(x)$ terminates within at most $p(|x|)$ steps (here, $|x|$ denotes the length of the string x). A probabilistic algorithm is one that has the capability of "tossing coins"; this is a metaphorical way of saying that the algorithm has access to a source of randomness that yields unbiased random bits that are each independently equal to 1 with probability $\frac{1}{2}$ and to 0 with probability $\frac{1}{2}$. Equivalently, we can view a randomized algorithm as one which is given, in addition to its input, a uniformly-distributed bit-string of "adequate length" on a special *random tape*. When considering a probabilistic polynomial-time algorithm with running time p and an input of length n, a random string of length $p(n)$ will certainly be adequate as the algorithm can only use $p(n)$ random bits within the allotted time.

Those familiar with complexity theory or algorithms will recognize that the idea of equating efficient computation with (probabilistic) polynomial-time computation is not unique to cryptography. The primary advantage of working with (probabilistic) polynomial-time algorithms is that this gives a class of algorithms that is closed under composition, meaning that a polynomial-time algorithm A that runs another polynomial-time algorithm A' as a sub-routine will also run in polynomial-time. Other than this useful fact, there is nothing inherently special about restricting adversaries to run in polynomial time, and essentially all the results we will see in this book could also be formulated in terms of adversaries running in, say, time $n^{\mathcal{O}(\log n)}$ (with honest parties still running in polynomial time).

Before proceeding, we address the question of why we consider *probabilistic* polynomial-time algorithms rather than just deterministic polynomial-time ones. There are two main answers for this. First, randomness is essential to cryptography (e.g., in order to choose random keys and so on) and so honest parties must be probabilistic. Given that this is the case, it is natural to consider adversaries that are probabilistic as well. A second reason for considering probabilistic algorithms is that the ability to toss coins may provide additional power. Since we use the notion of efficient computation to model realistic adversaries, it is important to make this class as large as possible (while still being realistic).

As an aside, we mention that the question of whether or not probabilistic polynomial-time adversaries are more powerful than deterministic polynomial-time adversaries is unresolved. In fact, recent results in complexity theory indicate that randomness does not help. Nevertheless, it does not hurt to model adversaries as probabilistic algorithms, and this can only provide stronger guarantees — that is, any scheme secure against probabilistic polynomial-time adversaries is certainly secure against deterministic polynomial-time adversaries as well.

Generating randomness. We have modeled all parties as probabilistic polynomial-time algorithms because, as we have mentioned, cryptography is only possible if randomness is available. (If secret keys cannot be generated at random, then an adversary can follow the same procedure used by the honest parties to obtain their "secret key". Recall that by Kerckhoffs' principle, we assume that this procedure is known.) Given this, one may wonder whether it is possible to actually "toss coins" on a computer and achieve probabilistic computation.

There are a number of ways "random bits" are obtained in practice. One solution is to use *hardware random number generators* that generate random bit-streams based on certain physical phenomena like thermal/electrical noise or radioactive decay. Another possibility is to use *software* random number generators which generate random bit-streams based on unpredictable behavior such as the time between key-strokes, movement of the mouse, hard disk access times, and so on. Some modern operating systems provide functions of this sort. In both of these cases, the underlying unpredictable event (whether natural or user-dependent) is unlikely to directly yield uniformly-distributed bits, and so further processing of the initial bit-stream is needed. Techniques for doing this are complex yet generally poorly understood, and are outside the scope of this text.

One must be careful in how random bits are chosen, and the use of badly-designed or inappropriate random number generators can often leave a good cryptosystem vulnerable to attack. Particular care must be taken to use a random number generator that is *designed for cryptographic use*, rather than a "general-purpose" random number generator which may be fine for some applications but not cryptographic ones. As one specific example, using the

`random()` function in the C programming language is a bad idea since it is not very random at all. Likewise, the current time (even to the millisecond) is not very random and cannot serve as the basis for a secret key.

Negligible Success Probability

Modern cryptography allows schemes that can be broken with very small probability to still be considered "secure". In the same way that we consider polynomial running times to be feasible, we consider inverse-polynomial probabilities to be significant. Thus, if an adversary could succeed in breaking a scheme with probability $1/p(n)$ for some (positive) polynomial p, then the scheme would not be considered secure. However, if the probability that the scheme can be broken is asymptotically smaller than $1/p(n)$ for *every* polynomial p, then we consider the scheme to be secure. This is due to the fact that the probability of adversarial success is so small that it is considered uninteresting. We call such probabilities of success *negligible*, and have the following definition.

DEFINITION 3.4 *A function f is* negligible *if for every polynomial $p(\cdot)$ there exists an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

An equivalent formulation of the above is to require that for all constants c there exists an N such that for all $n > N$ it holds that $f(n) < n^{-c}$. For shorthand, the above is also stated as follows: for every polynomial $p(\cdot)$ and *all sufficiently large values of* n it holds that $f(n) < \frac{1}{p(n)}$. This is, of course, the same. We typically denote an arbitrary negligible function by negl.

Example 3.5
The functions $2^{-n}, 2^{-\sqrt{n}}$,and $n^{-\log n}$ are all negligible. However, they approach zero at very different rates. In order to see this, we will show for what values of n each function is smaller than 10^{-6}:

1. $2^{20} = 1048576$ and thus for $n \geq 20$ we have that $2^{-n} < 10^{-6}$.
2. $2^{\sqrt{400}} = 1048576$ and thus for $n \geq 400$ we have that $2^{-\sqrt{n}} < 10^{-6}$.
3. $32^5 = 33554432$ and thus for $n \geq 32$ we have that $n^{-\log n} < 10^{-6}$.

From the above you may have the impression that $n^{-\log n}$ approaches zero more quickly than $2^{-\sqrt{n}}$. However this is incorrect; for all $n > 65536$ it holds that $2^{-\sqrt{n}} < n^{-\log n}$. Nevertheless, this does show that for values of n in the hundreds or thousands, an adversarial success probability of $n^{-\log n}$ is preferable to an adversarial success probability of $2^{-\sqrt{n}}$. ♢

A technical advantage of working with negligible success probabilities is that they also obey certain closure properties. The following is an easy exercise.

PROPOSITION 3.6 *Let* negl_1 *and* negl_2 *be negligible functions. Then,*

1. *The function* negl_3 *defined by* $\mathsf{negl}_3(n) = \mathsf{negl}_1(n) + \mathsf{negl}_2(n)$ *is negligible.*
2. *For any positive polynomial* p*, the function* negl_4 *defined by* $\mathsf{negl}_4(n) = p(n) \cdot \mathsf{negl}_1(n)$ *is negligible.*

The second part of the above proposition implies that if a certain event occurs with only negligible probability in a certain experiment, then the event occurs with negligible probability even if the experiment is repeated polynomially-many times. For example, the probability that n coin flips all come up "heads" is negligible. This means that even if we flip n coins polynomially-many times, the probability that *any* of these trials resulted in n heads is still negligible. (Formally this is proven using the union bound, stated in Proposition A.7.)

It is important to understand that events that occur with negligible probability can be safely ignored for all practical purposes (at least for large enough values of n). This is important enough to repeat and highlight:

> *Events that occur with negligible probability are so unlikely to occur that they can be ignored for all practical purposes. Therefore, a break of a cryptographic scheme that occurs with negligible probability is not significant.*

Lest you feel uncomfortable with the fact that an adversary can break a given scheme with some tiny (but non-zero) probability, note that with some tiny (but non-zero) probability the honest parties will be hit by an asteroid while executing the scheme! More benign, but making the same point: with some non-zero probability the hard drive of one of the honest parties will fail, thus erasing the secret key. Thus it simply does not make sense to worry about events that occur with sufficiently small probability.

Asymptotic Security: A Summary

Any security definition consists of two parts: a definition of what is considered a "break" of the scheme, and a specification of the power of the adversary. The power of the adversary can relate to many issues (e.g., in the case of encryption, whether we assume a ciphertext-only attack or a chosen-plaintext attack). However, when it comes to the *computational power* of the adversary, we will from now on model the adversary as efficient and thus probabilistic polynomial-time (meaning that only "feasible" adversarial strategies are considered). Definitions will also always be formulated so that a break that occurs with negligible probability is not considered significant. Thus, the general framework of any security definition will be as follows:

> A scheme is secure if for every *probabilistic polynomial-time adversary* $\mathcal{A}$ carrying out an attack of some specified type, the probability that $\mathcal{A}$ succeeds in this attack (where success is also well-defined) is *negligible*.

Such a definition is *asymptotic* because it is possible that for small values of n an adversary can succeed with high probability. In order to see this in more detail, we will use the full definition of "negligible" in the above statement:

> A scheme is secure if for every probabilistic polynomial-time adversary $\mathcal{A}$ carrying out an attack of some specified type, and for every polynomial $p(\cdot)$, there exists an integer N such that the probability that $\mathcal{A}$ succeeds in this attack is less than $\frac{1}{p(n)}$ for every $n > N$.

Note that nothing is guaranteed for values $n \leq N$.

3.1.3 Proofs by Reduction

As we have seen, a cryptographic scheme that is computationally secure (but not perfectly secure) can always be broken given enough time. To prove *unconditionally* that some scheme is computationally secure would thus require proving a lower bound on the time needed to break the scheme. Specifically, it would be necessary to prove that the scheme cannot be broken by any polynomial-time algorithm. Unfortunately, the current state of affairs is such that we are unable to prove lower bounds of this type. In fact, an unconditional proof of security for any modern encryption scheme would require breakthrough results in complexity theory that seem far out of reach today.[4] This might seem to leave us with no choice but to simply assume that a given scheme is secure. As discussed in Section 1.4, however, this is a very undesirable approach and one that history has taught us is very dangerous.

Instead of blithely assuming that a given cryptographic construction is secure, our strategy instead will be to assume that some *low-level* problem is hard to solve, and then to *prove* that the construction in question is secure under this assumption. In Section 1.4.2 we have already explained in great detail why this approach is preferable so we do not repeat those arguments here.

The proof that a given cryptographic construction is secure as long as some underlying problem is hard generally proceeds by presenting an explicit *reduction* showing how to convert any efficient adversary $\mathcal{A}$ that succeeds in "breaking" the construction with non-negligible probability into an efficient algorithm $\mathcal{A}'$ that succeeds in solving the problem that was assumed to be

[4] For those familiar with the $\mathcal{P}$ versus $\mathcal{NP}$ question, we remark that an unconditional proof of security for any encryption scheme in which messages are longer than the key would imply a proof that $\mathcal{P} \neq \mathcal{NP}$.

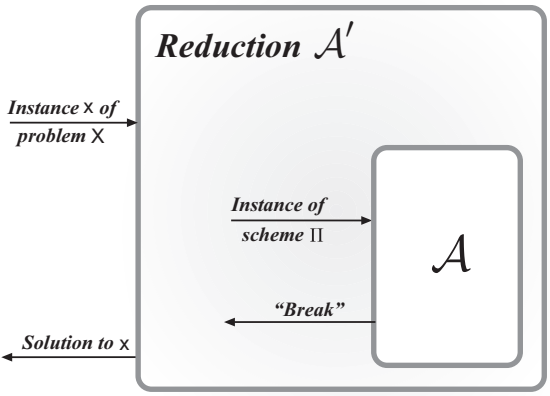

FIGURE 3.1: A high-level overview of a security proof by reduction.

hard. (In fact, this is the only sort of proof we use in this book.) Since this is so important, we walk through a high-level outline of the steps of such a proof in detail. We begin with an assumption that some problem X cannot be solved (in some precisely-defined sense) by any polynomial-time algorithm except with negligible probability. We want to prove that some cryptographic construction Π is secure (again, in some sense that is precisely defined). To do this:

1. Fix some efficient (i.e., probabilistic polynomial-time) adversary $\mathcal{A}$ attacking Π. Denote this adversary's success probability by $\varepsilon(n)$.

2. Construct an efficient algorithm $\mathcal{A}'$, called the "reduction" that attempts to solve problem X using adversary $\mathcal{A}$ as a sub-routine. An important point here is that $\mathcal{A}'$ knows nothing about "how" $\mathcal{A}$ works; the only thing $\mathcal{A}'$ knows is that $\mathcal{A}$ is expecting to attack Π. So, given some input instance x of problem X, our algorithm $\mathcal{A}'$ will *simulate* for $\mathcal{A}$ an instance of Π such that:

   (a) As far as $\mathcal{A}$ can tell, it is interacting with Π. More formally, the view of $\mathcal{A}$ when it is run as a sub-routine by $\mathcal{A}'$ should be distributed identically to (or at least close to) the view of $\mathcal{A}$ when it interacts with Π itself.

   (b) If $\mathcal{A}$ succeeds in "breaking" the instance of Π that is being simulated by $\mathcal{A}'$, this should allow $\mathcal{A}'$ to solve the instance x it was given, at least with inverse polynomial probability $1/p(n)$.

3. Taken together, 2(a) and 2(b) imply that *if* $\varepsilon(n)$ is not negligible, *then* $\mathcal{A}'$ solves problem X with non-negligible probability $\varepsilon(n)/p(n)$. Since $\mathcal{A}'$ is efficient, and runs the PPT adversary $\mathcal{A}$ as a sub-routine, this implies an efficient algorithm solving X with non-negligible probability, contradicting the initial assumption.

4. We conclude that, given the assumption regarding X, *no* efficient adversary $\mathcal{A}$ can succeed in breaking Π with probability that is not negligible. Stated differently, Π is computationally secure.

This will become more clear when we see examples of such proofs in the sections that follow.

3.2 Defining Computationally-Secure Encryption

Given the background of the previous section, we are ready to present a definition of computational security for private-key encryption. First, we redefine the *syntax* of private-key encryption; this will essentially be the same as the syntax introduced in Chapter 2 except that we will now explicitly take into account the security parameter. We will also now let the message space be, by default, the set $\{0,1\}^*$ of all (finite-length) binary strings.

DEFINITION 3.7 *A* private-key encryption scheme *is a tuple of probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *such that:*

1. *The* key-generation algorithm Gen *takes as input the security parameter* 1^n *and outputs a key* k*; we write this as* $k \leftarrow \mathsf{Gen}(1^n)$ *(thus emphasizing the fact that* Gen *is a randomized algorithm). We will assume without loss of generality that any key* k *output by* $\mathsf{Gen}(1^n)$ *satisfies* $|k| \geq n$*.*

2. *The* encryption algorithm Enc *takes as input a key* k *and a plaintext message* $m \in \{0,1\}^*$*, and outputs a ciphertext* c*.*[5] *Since* Enc *may be randomized, we write this as* $c \leftarrow \mathsf{Enc}_k(m)$*.*

3. *The* decryption algorithm Dec *takes as input a key* k *and a ciphertext* c*, and outputs a message* m*. We assume that* Dec *is deterministic, and so write this as* $m := \mathsf{Dec}_k(c)$*.*

It is required that for every n*, every key* k *output by* $\mathsf{Gen}(1^n)$*, and every* $m \in \{0,1\}^*$*, it holds that* $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m$*.*[6]

If $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is such that for* k *output by* $\mathsf{Gen}(1^n)$*, algorithm* Enc_k *is only defined for messages* $m \in \{0,1\}^{\ell(n)}$*, then we say that* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a* fixed-length *private-key encryption scheme for messages of length* $\ell(n)$*.*

[5] As a technical condition, Enc is allowed to run in time polynomial in $|k| + |m|$ (i.e., the total length of its inputs). If we only included $|m|$ then encrypting a single bit would not take polynomial time, and if we only included $|k|$ then we would have to a priori bound the length of messages m that could be encrypted. Although needed, this technicality can be ignored from here on.

[6] Given this, our assumption that Dec is deterministic is without loss of generality.

We remark that it is almost always the case in this chapter and the next that $\mathsf{Gen}(1^n)$ chooses $k \leftarrow \{0,1\}^n$ uniformly at random.

3.2.1 The Basic Definition of Security

There are a number of standard ways of defining security for private-key encryption, with the main differences relating to the assumed power of the adversary in its attack. We begin by presenting the most basic notion of security — security against a weak form of ciphertext-only attack where the adversary only observes a *single* ciphertext — and consider stronger definitions of security later in the chapter.

Motivating the definition. As discussed in Chapter 1, any definition of security consists of two distinct components: a specification of the assumed power of the adversary, and a description of what constitutes a "break" of the scheme. We begin our definitional treatment by considering the case of an *eavesdropping adversary* who observes the encryption of a single message or, equivalently, is given a single ciphertext that it wishes to "crack". This is a rather weak class of adversaries, but is exactly the type of adversary that was considered in the previous chapter (we will encounter stronger adversaries later in the chapter). Of course, as explained in the previous section, we are now interested only in adversaries that are computationally bounded and limited to running in polynomial time.

Although we have made two substantial assumptions about the adversary's capabilities (i.e., that it only eavesdrops, and that it runs in polynomial time), we make no assumptions whatsoever about the adversary's *strategy*. This is crucial for obtaining meaningful notions of security because it is impossible to predict all possible strategies. We therefore protect against all strategies that can be carried out by adversaries within the class we have defined.

Defining the "break" for encryption is not trivial, but we have already discussed this issue at length in Section 1.4.1 and in the previous chapter. We therefore just recall that the idea behind the definition is that the adversary should be unable to learn *any partial information* about the plaintext from the ciphertext. The definition of *semantic security* directly formalizes exactly this notion, and was the first definition of security for encryption to be proposed. Unfortunately, the definition of semantic security is complex and difficult to work with. Fortunately, there is an equivalent definition in terms of *indistinguishability* which is much simpler. Since the definitions are equivalent, we can work with the simpler definition of indistinguishability while being convinced that the security guarantees we obtain are those we expect from semantic security. (See Section 3.2.2 for further discussion on this point.)

The definition of indistinguishability we give here is syntactically almost identical to the alternative definition of perfect secrecy given as Definition 2.4. (This serves as further motivation that the definition of indistinguishability is a "good" one.) Recall that Definition 2.4 considers an experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$

in which an adversary $\mathcal{A}$ outputs two messages m_0 and m_1, and is then given an encryption of one of these messages, chosen at random, using a randomly-generated key. The definition states that a scheme Π is secure if no adversary $\mathcal{A}$ can determine which of the messages m_0, m_1 was encrypted with probability any different from $1/2$ (which is the probability of just making a random guess).

Here, we keep the experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$ almost exactly the same (except for some technical differences discussed below), but introduce two key modifications in the definition itself:

1. We now consider only adversaries running in *polynomial time*, whereas Definition 2.4 considered even all-powerful adversaries.

2. We now concede that the adversary might determine the encrypted message with probability *negligibly better than* $1/2$.

As discussed extensively in the previous section, the above relaxations constitute the core elements of computational security.

As for the differences introduced in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$, one is purely syntactic while the other is introduced for technical reasons. The most prominent difference is that we now parameterize the experiment by a security parameter n. We then measure both the running time of the adversary $\mathcal{A}$ as well as its success probability as functions of n. We write $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ to denote the experiment being run with the given value of the security parameter, and write

$$\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] \tag{3.1}$$

to denote the probability that $\mathcal{A}$ outputs 1 in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. It is important to note that when $\mathcal{A}$ and Π are fixed, Equation (3.1) is a function of n.

The second difference in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$ is that we now require the adversary to output two messages m_0, m_1 *of equal length*. From a theoretical point of view, this restriction is necessary because of our requirement that an encryption scheme should be able to encrypt arbitrary-length messages. This restriction could be removed if we were willing to forego this requirement, as we did in the case of perfect secrecy; see Exercises 3.3 and 3.4 for more on this issue. This restriction is also appropriate for most encryption schemes used in practice, where different-length messages result in different-length ciphertexts, and so an adversary could trivially distinguish which message was encrypted if it were allowed to output messages of different lengths.

Most encryption schemes used in practice do *not* hide the length of messages that are encrypted. In cases where the length of a message might itself represent sensitive information (e.g., when it indicates the number of digits in an employee's salary), care must be taken to *pad* the input to some fixed length before encrypting. We do not discuss this further.

Indistinguishability in the presence of an eavesdropper. We now give the formal definition, beginning with the experiment outlined above. The experiment is defined for any private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, any adversary $\mathcal{A}$, and any value n for the security parameter:

The eavesdropping indistinguishability experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$:

1. *The adversary $\mathcal{A}$ is given input 1^n, and outputs a pair of messages m_0, m_1 of the same length.*
2. *A key k is generated by running $\mathsf{Gen}(1^n)$, and a random bit $b \leftarrow \{0,1\}$ is chosen. A ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$. We call c the* challenge ciphertext.
3. *$\mathcal{A}$ outputs a bit b'.*
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. If $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$, we say that $\mathcal{A}$* succeeded.

There is no limitation on the length of the messages m_0 and m_1 to be encrypted, as long as they are the same. (Of course, since the adversary is restricted to run in polynomial time, m_0 and m_1 have length polynomial in n.) If Π is a fixed-length scheme for messages of length $\ell(n)$, the above experiment is modified by requiring $m_0, m_1 \in \{0,1\}^{\ell(n)}$.

The definition of indistinguishability states that an encryption scheme is secure if the success probability of any PPT adversary in the above experiment is at most negligibly greater than 1/2. (Note that it is easy to succeed with probability 1/2 by just outputting a random bit b'. The challenge is to do better than this.) We are now ready for the definition.

DEFINITION 3.8 *A private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ has* indistinguishable encryptions in the presence of an eavesdropper *if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over the random coins used by $\mathcal{A}$, as well as the random coins used in the experiment (for choosing the key, the random bit b, and any random coins used in the encryption process).

The definition quantifies over *all* probabilistic polynomial-time adversaries, meaning that security is required for all "feasible" strategies (where we equate feasible strategies with those that can be carried out in polynomial time). The fact that the adversary has only eavesdropping capabilities is implicit in the fact that its input is limited to a (single) ciphertext, and the adversary does not have any further interaction with the sender or the receiver. (As we will

see later, allowing additional interaction results in a significantly stronger adversary.) Now, the definition states simply that any adversary $\mathcal{A}$ will succeed in guessing which message was encrypted with probability at most negligibly better than a naive guess (which is correct with probability 1/2).

Notice that the adversary is allowed to choose the messages m_0 and m_1. Thus, even though it knows that c is an encryption of one of these plaintext messages, it still cannot determine which one was encrypted. This is a very strong guarantee, and one that has great practical importance. Consider, for example, a scenario whereby the adversary knows that the message being encrypted is either "attack today" or "don't attack." Clearly, we do not want the adversary to know which message was encrypted, even though it already knows that it is one of these two possibilities.

An equivalent formulation. Definition 3.8 states that an eavesdropping adversary cannot determine which plaintext was encrypted with probability significantly better than it could achieve by taking a random guess. An equivalent way of formalizing the definition is to state that every adversary *behaves the same way* whether it sees an encryption of m_0 or an encryption of m_1 (for any m_0, m_1 of the same length). Since $\mathcal{A}$ outputs a single bit, "behaving the same way" means that it outputs 1 with almost the same probability in each case. To formalize this, define $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n,b)$ to be as above, except that the fixed bit b is used (rather than being chosen at random). In addition, denote the output bit b' of $\mathcal{A}$ in $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n,b)$ by $\mathsf{output}(\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n,b))$. The following definition essentially states that $\mathcal{A}$ cannot determine whether it is running in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n,0)$ or experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n,1)$.

DEFINITION 3.9 *A private-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable encryptions in the presence of an eavesdropper *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\left|\Pr[\mathsf{output}(\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n,0)) = 1] - \Pr[\mathsf{output}(\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n,1)) = 1]\right| \leq \mathsf{negl}(n).$$

The fact that this definition is equivalent to Definition 3.8 is left as an exercise.

3.2.2 * Properties of the Definition

We motivated the definition of secure encryption by saying that it should be infeasible for an adversary to learn any partial information about the plaintext from the ciphertext. However, the actual definition of indistinguishability looks very different. As we have mentioned, Definition 3.8 is indeed equivalent to *semantic security* that formalizes the intuitive notion that partial information cannot be learned. We will not prove full equivalence here. Instead, we will prove two claims demonstrating that indistinguishability implies weaker

versions of semantic security. For comparison, we then present (essentially) the full definition of semantic security. The reader is referred to [65, Chapter 5.2] for further discussion and a full proof of equivalence.

We begin by showing that indistinguishability implies that no single bit of a *randomly chosen* plaintext can be guessed with probability significantly better than 1/2. Below, we denote by m^i the ith bit of m, and set $m^i = 0$ if $i > |m|$.

CLAIM 3.10 *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper. Then for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *and all* i*, there exists a negligible function* negl *such that:*

$$\Pr\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m)) = m^i\right] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where m *is chosen uniformly at random from* $\{0,1\}^n$*, and the probability is taken over the random coins of* $\mathcal{A}$*, the choice of* m *and the key* k*, and any random coins used in the encryption process.*

PROOF The idea behind the proof of this claim is that if it is possible to guess the ith bit of m given $\mathsf{Enc}_k(m)$, then it is also possible to distinguish between encryptions of plaintext messages m_0 and m_1 where the ith bit of m_0 equals 0 and the ith bit of m_1 equals 1. Specifically, given a ciphertext c try to compute the ith bit of the underlying plaintext. If this computation indicates that the ith bit is 0, then guess that m_0 was encrypted; if it indicates that the ith bit is 1, then guess that m_1 was encrypted. Formally, we show that if there exists an adversary $\mathcal{A}$ that can guess the ith bit of m given $\mathsf{Enc}_k(m)$ with probability at least $1/2 + \varepsilon(n)$ for some function $\varepsilon(\cdot)$, then there exists an adversary that succeeds in the indistinguishability experiment for $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with probability $1/2 + \varepsilon(n)$. If Π has indistinguishable encryptions, then $\varepsilon(\cdot)$ must be negligible.

In detail, let $\mathcal{A}$ be a probabilistic polynomial-time adversary and define $\varepsilon(\cdot)$ as follows:

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m)) = m^i\right] - \frac{1}{2},$$

where m is chosen uniformly from $\{0,1\}^n$. From now on, for visual clarity, we no longer explicitly indicate the input 1^n to $\mathcal{A}$. Take $n \geq i$, let I_0^n be the set of all strings of length n whose ith bit is 0, and let I_1^n be the set of all strings of length n whose ith bit is 1. It follows that:

$$\Pr\left[\mathcal{A}(\mathsf{Enc}_k(m)) = m^i\right] = \frac{1}{2} \cdot \Pr\left[\mathcal{A}(\mathsf{Enc}_k(m_0)) = 0\right] + \frac{1}{2} \cdot \left[\mathcal{A}(\mathsf{Enc}_k(m_1)) = 1\right],$$

where m_0 is chosen uniformly from I_0^n and m_1 is chosen uniformly from I_1^n. (The above holds because I_0^n and I_1^n each contain exactly half of $\{0,1\}^n$. Therefore, the probability that m lies in each set is exactly 1/2.)

Consider now the following adversary $\mathcal{A}'$ who eavesdrops on the encryption of a single message:

Adversary $\mathcal{A}'$:

1. On input 1^n (with $n \geq i$), choose $m_0 \leftarrow I_0^n$ and $m_1 \leftarrow I_1^n$ uniformly at random from the indicated sets. Output m_0, m_1.
2. Upon receiving a ciphertext c, invoke $\mathcal{A}$ on input c. Output $b' = 0$ if $\mathcal{A}$ outputs 0, and $b' = 1$ if $\mathcal{A}$ outputs 1.

$\mathcal{A}'$ runs in polynomial time since $\mathcal{A}$ does.

Using the definition of experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n)$ (for $n \geq i$), note that $b' = b$ if and only if $\mathcal{A}$ outputs b upon receiving $\mathsf{Enc}_k(m_b)$. So

$$\begin{aligned}
\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1\right] &= \Pr\left[\mathcal{A}(\mathsf{Enc}_k(m_b)) = b\right] \\
&= \frac{1}{2} \cdot [\mathcal{A}(\mathsf{Enc}_k(m_0)) = 0] + \frac{1}{2} \cdot [\mathcal{A}(\mathsf{Enc}_k(m_1)) = 1] \\
&= \Pr\left[\mathcal{A}(\mathsf{Enc}_k(m)) = m^i\right] \\
&= \frac{1}{2} + \varepsilon(n).
\end{aligned}$$

By the assumption that $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ has indistinguishable encryptions in the presence of an eavesdropper, it follows that $\varepsilon(\cdot)$ must be negligible. (Note that it does not matter what happens when $n < i$, since we are concerned with asymptotic behavior only.) This completes the proof. ∎

We now proceed to show, roughly speaking, that no PPT adversary can learn *any* function of the plaintext message given the ciphertext, and furthermore that this holds regardless of the *a priori* distribution over the message being sent. This requirement is non-trivial to define formally. To see why, note that even for the case of computing the ith bit of the plaintext message m (as considered above), it is very easy to compute this value if m is chosen, for example, uniformly from the set I_0^n (rather than uniformly from $\{0,1\}^n$). Thus, what we actually want to say is that if an adversary receiving the ciphertext $c = \mathsf{Enc}_k(m)$ can compute $f(m)$ for some function f, then there exists an adversary that can compute $f(m)$ with the same probability of being correct, but *without* being given the ciphertext (and only knowing the *a priori* distribution on m).

In the next claim we show the above when m is chosen uniformly at random from some set $S \subseteq \{0,1\}^n$. Thus, if the plaintext is an email message, we can take S to be the set of English-language messages with correct email headers. Actually, since we are considering an asymptotic setting, we will work with an infinite set $S \subseteq \{0,1\}^*$. Then for security parameter n, a plaintext message is chosen uniformly from $S_n \stackrel{\text{def}}{=} S \cap \{0,1\}^n$ (i.e., the subset of strings of S having length n), which is assumed never to be empty. As a

technical condition, we also need to assume that it is possible to efficiently sample strings uniformly from S_n; that is, that there exists some probabilistic polynomial-time algorithm that, on input 1^n, outputs a uniform element of S_n. We refer to this by saying that the set S is *efficiently sampleable*. We also restrict to functions f that can be computed in polynomial time.

CLAIM 3.11 *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper. Then for every probabilistic polynomial-time adversary* $\mathcal{A}$ *there exists a probabilistic polynomial-time algorithm* $\mathcal{A}'$ *such that for all polynomial-time computable functions* f *and all efficiently-sampleable sets* S*, there exists a negligible function* negl *such that:*

$$\left| \Pr\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m)) = f(m)\right] - \Pr\left[\mathcal{A}'(1^n) = f(m)\right] \right| \leq \mathsf{negl}(n),$$

where m *is chosen uniformly at random from* $S_n \stackrel{\text{def}}{=} S \cap \{0,1\}^n$*, and the probabilities are taken over the choice of* m *and the key* k*, and any random coins used by* $\mathcal{A}$*,* $\mathcal{A}'$*, and the encryption process.*

PROOF (Sketch) We present only an informal sketch of the proof of this claim. Assume that $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ has indistinguishable encryptions. This implies that no probabilistic polynomial-time adversary $\mathcal{A}$ can distinguish between $\mathsf{Enc}_k(m)$ and $\mathsf{Enc}_k(1^n)$, for any $m \in \{0,1\}^n$. Consider now the probability that $\mathcal{A}$ successfully computes $f(m)$ given $\mathsf{Enc}_k(m)$. We claim that $\mathcal{A}$ should successfully compute $f(m)$ given $\mathsf{Enc}_k(1^n)$ with almost the same probability. Otherwise, $\mathcal{A}$ could be used to distinguish between $\mathsf{Enc}_k(m)$ and $\mathsf{Enc}_k(1^n)$. (The distinguisher is easily constructed: choose $m \leftarrow S_n$ uniformly at random — here we use the assumption that S is efficiently-sampleable — and output $m_0 = m$, $m_1 = 1^n$. When given a ciphertext c that is either an encryption of m or 1^n, invoke $\mathcal{A}$ on c, and output 0 if and only if $\mathcal{A}$ outputs $f(m)$. If $\mathcal{A}$ outputs $f(m)$ when it is given an encryption of m with probability that is non-negligibly different than when it is given an encryption of 1^n, then the described distinguisher violates Definition 3.9.)

The above observation yields the following algorithm $\mathcal{A}'$ that does not receive $c = \mathsf{Enc}_k(m)$, yet computes $f(m)$ equally well: on input the security parameter 1^n, $\mathcal{A}'$ chooses a random key k, invokes $\mathcal{A}$ on $c \leftarrow \mathsf{Enc}_k(1^n)$, and outputs whatever $\mathcal{A}$ outputs. By the above, we have that $\mathcal{A}$ outputs $f(m)$ when run as a sub-routine by $\mathcal{A}'$ with almost the same probability as when it receives $\mathsf{Enc}_k(m)$. Thus, $\mathcal{A}'$ fulfills the property required by the claim. ■

Semantic security. The full definition of semantic security is considerably more general than the property proven in Claim 3.11. Arbitrary distributions over plaintext messages are now considered, and the definition additionally takes into account arbitrary "external" information about the plaintext that

may be "leaked" to the adversary through other means (e.g., because the same message m is used for other purposes as well). As above, we denote by f the function of the plaintext that the adversary is attempting to compute. Rather than consider a specific set S (and a uniform distribution over subsets of S), we consider an arbitrary distribution $X = (X_1, X_2, \ldots)$, where, for security parameter n, the plaintext is chosen according to distribution X_n. We require that X be efficiently sampleable, implying here the existence of a PPT algorithm that, on input 1^n, outputs an element chosen according to distribution X_n. We also require that, for all n, all strings in X_n have the same length. Finally, we model "external" information about m that is "leaked" to the adversary by giving the adversary $h(m)$, for some arbitrary function h, in addition to an encryption of m. That is:

DEFINITION 3.12 *A private-key encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* semantically secure in the presence of an eavesdropper *if for every probabilistic polynomial-time algorithm $\mathcal{A}$ there exists a probabilistic polynomial-time algorithm $\mathcal{A}'$ such that for all efficiently-sampleable distributions $X = (X_1, \ldots)$ and all polynomial-time computable functions f and h, there exists a negligible function* negl *such that*

$$\left| \Pr[\mathcal{A}(1^n, \mathsf{Enc}_k(m), h(m)) = f(m)] - \Pr[\mathcal{A}'(1^n, h(m)) = f(m)] \right| \leq \mathsf{negl}(n),$$

where m is chosen according to distribution X_n, and the probabilities are taken over the choice of m and the key k, and any random coins used by $\mathcal{A}$, $\mathcal{A}'$, and the encryption process.

The adversary $\mathcal{A}$ is given the ciphertext $\mathsf{Enc}_k(m)$ as well as the history function $h(m)$, where this latter function represents whatever "external" knowledge of the plaintext m the adversary may have. The adversary $\mathcal{A}$ then attempts to guess the value of $f(m)$. Algorithm $\mathcal{A}'$ also attempts to guess the value of $f(m)$, but is given *only* $h(m)$. The security requirement states that $\mathcal{A}$'s success in guessing $f(m)$, when given the ciphertext, can be essentially matched by some algorithm $\mathcal{A}'$ who is not given the ciphertext. Thus, the ciphertext $\mathsf{Enc}_k(m)$ does not reveal anything new about the value of $f(m)$.

Definition 3.12 constitutes a very strong and convincing formulation of the security guarantees that should be provided by an encryption scheme. Arguably, it is much more convincing than indistinguishability (that only considers two plaintexts, and does not mention external knowledge or arbitrary distributions). However, it is technically easier to work with the definition of indistinguishability (e.g., for proving that a given scheme is secure). Fortunately, it has been shown that the definitions are *equivalent*:

THEOREM 3.13 *A private-key encryption scheme has indistinguishable encryptions in the presence of an eavesdropper if and only if it is semantically secure in the presence of an eavesdropper.*

Looking ahead, a similar equivalence is known for all the definitions of indistinguishability that we present in this chapter. We can therefore use indistinguishability as our working definition, while being assured that the security guarantees achieved are those of semantic security.

3.3 Pseudorandomness

Having defined what it means for an encryption scheme to be secure, the reader may expect us to begin immediately with constructions of secure encryption schemes. However, before doing so we need to introduce the notion of pseudorandomness. This notion plays a fundamental role in cryptography in general, and private-key encryption in particular. Loosely speaking, a pseudorandom string is a string that looks like a uniformly distributed string, as long as the entity that is "looking" runs in polynomial time. Just as indistinguishability can be viewed as a computational relaxation of perfect secrecy, pseudorandomness is a computational relaxation of true randomness.

An important conceptual point is that, technically speaking, no fixed string can be said to be "pseudorandom" (in the same way that it does not make much sense to refer to any fixed string as "random"). Rather, pseudorandomness actually refers to a *distribution* on strings, and when we say that a distribution $\mathcal{D}$ over strings of length ℓ is pseudorandom this means that $\mathcal{D}$ is indistinguishable from the uniform distribution over strings of length ℓ. (Strictly speaking, since we are in an asymptotic setting we actually need to speak of the pseudorandomness of a *sequence* of distributions $\mathcal{D} = \{\mathcal{D}_n\}$, where distribution $\mathcal{D}_n$ is associated with security parameter n. We ignore this point in our current discussion.) More precisely, it is infeasible for any polynomial-time algorithm to tell whether it is given a string sampled according to $\mathcal{D}$ or an ℓ-bit string chosen uniformly at random.

Even given the above discussion, we frequently abuse notation and call a string sampled according to the uniform distribution a "random string", and a string sampled according to a pseudorandom distribution $\mathcal{D}$ a "pseudorandom string". This is only useful shorthand.

Before proceeding, we provide some intuition as to why pseudorandomness helps in the construction of secure private-key encryption schemes. On a simplistic level, if a ciphertext looks random, then it is clear that no adversary can learn any information from it about the plaintext. To some extent, this is the exact intuition that lies behind the perfect secrecy of the one-time pad. In that case, the ciphertext is uniformly distributed (assuming the key is unknown) and thus reveals nothing about the plaintext. (Of course, such statements only appeal to intuition and do not constitute a formal argument.) The one-time pad worked by computing the XOR of a random string (the key)

with the plaintext. If a *pseudorandom* string were used instead, this should not make any noticeable difference to a polynomial-time observer. Thus, security should still hold for polynomial-time adversaries.

As we will see below, this idea can be implemented. The advantage of using a pseudorandom string rather than a truly random string is that a long pseudorandom string can be generated from a relatively short random seed (or key). Thus, a short key can be used to encrypt a long message, something that is impossible when perfect secrecy is required.

Pseudorandom generators. Informally, as discussed above, a distribution $\mathcal{D}$ is pseudorandom if no polynomial-time distinguisher can detect if it is given a string sampled according to $\mathcal{D}$ or a string chosen uniformly at random. Similarly to Definition 3.9, this is formalized by requiring that every polynomial-time algorithm outputs 1 with almost the same probability when given a truly random string and when given a pseudorandom one (this output bit is interpreted as the algorithm's "guess"). A pseudorandom generator is a *deterministic* algorithm that receives a short truly random seed and stretches it into a long string that is pseudorandom. Stated differently, a pseudorandom generator uses a small amount of true randomness in order to generate a large amount of pseudorandomness. In the definition that follows, we set n to be the length of the seed that is input to the generator and $\ell(n)$ to be the output length. Clearly, the generator is only interesting if $\ell(n) > n$ (otherwise, it doesn't generate any new "randomness").

DEFINITION 3.14 *Let $\ell(\cdot)$ be a polynomial and let G be a deterministic polynomial-time algorithm such that for any input $s \in \{0,1\}^n$, algorithm G outputs a string of length $\ell(n)$. We say that G is a* pseudorandom generator *if the following two conditions hold:*

1. (Expansion:) *For every n it holds that $\ell(n) > n$.*
2. (Pseudorandomness:) *For all probabilistic polynomial-time distinguishers D, there exists a negligible function* negl *such that:*

$$\left|\Pr[D(r)=1] - \Pr[D(G(s))=1]\right| \leq \mathsf{negl}(n),$$

where r is chosen uniformly at random from $\{0,1\}^{\ell(n)}$, the seed *s is chosen uniformly at random from $\{0,1\}^n$, and the probabilities are taken over the random coins used by D and the choice of r and s.*

The function $\ell(\cdot)$ is called the expansion factor *of G.*

Discussion. We stress that the output of a pseudorandom generator is actually very *far* from random. To see this, consider the case that $\ell(n) = 2n$ and so G doubles the length of its input. The uniform distribution over $\{0,1\}^{2n}$ is characterized by the fact that each of the 2^{2n} possible strings is chosen with

probability exactly 2^{-2n}. In contrast, consider the distribution generated by G. Since G receives an input of length n, the number of different possible strings in its range is at most 2^n. Thus, the probability that a random string of length $2n$ is in the range of G is at most $2^n/2^{2n} = 2^{-n}$ (just take the total number of strings in the range of G and divide it by the number of strings of length $2n$). That is, most strings of length $2n$ do not occur as outputs of G.

This in particular means that it is trivial to distinguish between a random string and a pseudorandom string *given an unlimited amount of time*. Consider the following exponential-time D that works as follows: upon input some string w, distinguisher D outputs 1 if and only if there exists an $s \in \{0,1\}^n$ such that $G(s) = w$. (This computation is carried out by searching all of $\{0,1\}^n$ and computing $G(s)$ for every $s \in \{0,1\}^n$. This computation can be carried out because the specification of G is known; only its random seed is unknown.) Now, if w was generated by G, it holds that D outputs 1 with probability 1. In contrast, if w is uniformly distributed in $\{0,1\}^{2n}$ then the probability that there exists an s with $G(s) = w$ is at most 2^{-n}, and so D outputs 1 in this case with probability at most 2^{-n}. We therefore have that

$$\left| \Pr[D(r) = 1] - \Pr[D(G(s)) = 1] \right| = 1 - 2^{-n},$$

which is large. This type of attack is called a brute force attack because it just tries all possible seeds. The advantage of such an "attack" is that it is applicable to all generators, irrespective of how they work.

The above shows that the distribution generated by G is not the uniform one. Nevertheless, *polynomial-time distinguishers* do not have time to carry out the above procedure. Indeed, if G is a pseudorandom generator, then it is guaranteed that there do not exist *any* polynomial-time procedures that reliably distinguish random and pseudorandom strings. This means that *pseudorandom strings are just as good as truly random ones*, as long as the seed is kept secret and we are considering only polynomial-time observers.

The seed and its length. The seed for a pseudorandom generator must be chosen uniformly at random, and be kept entirely secret from the distinguisher. Another important point, evident from the above discussion of brute-force attacks, is that s must be long enough so that no "efficient algorithm" has time to traverse all possible seeds. Technically, this is taken care of by the fact that all algorithms are assumed to run in polynomial time and thus cannot search through all 2^n possible seeds when n is large enough. In practice, however, the seed must be taken to be of some concrete length. Based on the above, *at the very least* s must be long enough so that it is impossible to efficiently try all possible seeds.

Existence of pseudorandom generators. The first question one should ask is whether any entity satisfying Definition 3.14 even exists. Unfortunately, we do not know how to unequivocally prove the existence of pseudorandom generators. Nevertheless, we believe that pseudorandom generators exist, and

this belief is based on the fact that they can be constructed (in a provable sense) under the rather weak assumption that *one-way functions* exist. This will be discussed in greater detail in Chapter 6. For now, it suffices to say that there are certain long-studied problems that have no known efficient algorithm and that are widely assumed to be unsolvable in polynomial-time. An example of such a problem is integer factorization: i.e., the problem of finding the prime factors of a given number. What is important for our discussion here is that one-way functions, and hence pseudorandom generators, can be constructed under the assumption that these problems really are "hard".

In practice, various constructions believed to act as pseudorandom generators are known. In fact, as we will see later in this chapter and in Chapter 5, constructions exist that are believed to satisfy even stronger requirements.

3.4 Constructing Secure Encryption Schemes

3.4.1 A Secure Fixed-Length Encryption Scheme

We are now ready to construct a fixed-length encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper. The encryption scheme we construct is very similar to the one-time pad encryption scheme (see Section 2.2), except that a *pseudorandom* string is used as the "pad" rather than a random string. Since a pseudorandom string "looks random" to any polynomial-time adversary, the encryption scheme can be proven to be computationally-secure.

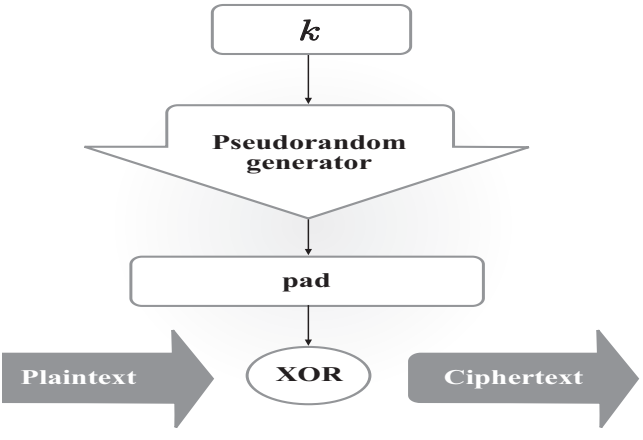

FIGURE 3.2: Encryption with a pseudorandom generator.

The encryption scheme. Let G be a pseudorandom generator with expansion factor ℓ (that is, $|G(s)| = \ell(|s|)$). Recall that an encryption scheme is defined by three algorithms: a key-generation algorithm Gen, an encryption algorithm Enc, and a decryption algorithm Dec. The encryption process works by applying a pseudorandom generator to the key (which serves as a seed) in order to obtain a long pad that is then XORed to the plaintext message. The scheme is formally described in Construction 3.15, and is depicted graphically in Figure 3.2.

CONSTRUCTION 3.15

Let G be a pseudorandom generator with expansion factor ℓ. Define a private-key encryption scheme for messages of length ℓ as follows:

- Gen: on input 1^n, choose $k \leftarrow \{0,1\}^n$ uniformly at random and output it as the key.
- Enc: on input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^{\ell(n)}$, output the ciphertext
$$c := G(k) \oplus m.$$
- Dec: on input a key $k \in \{0,1\}^n$ and a ciphertext $c \in \{0,1\}^{\ell(n)}$, output the plaintext message
$$m := G(k) \oplus c.$$

A private-key encryption scheme from any pseudorandom generator.

We now prove that the given encryption scheme has indistinguishable encryptions in the presence of an eavesdropper, under the *assumption* that G is a pseudorandom generator. Notice that our claim is not unconditional. Rather, we *reduce* the security of the encryption scheme to the properties of G as a pseudorandom generator. This is a very important proof technique that was described in Section 3.1.3 and will be discussed further after the proof itself.

THEOREM 3.16 *If G is a pseudorandom generator, then Construction 3.15 is a fixed-length private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper.*

PROOF Let Π denote Construction 3.15. We show that if there exists a probabilistic polynomial-time adversary $\mathcal{A}$ for which Definition 3.8 does not hold, then we can construct a probabilistic polynomial-time algorithm that distinguishes the output of G from a truly random string. The intuition behind this claim is that if Π used a truly random string in place of the pseudorandom string $G(k)$, then the resulting scheme would be identical to the one-time pad encryption scheme and $\mathcal{A}$ would be unable to correctly guess which message was encrypted with probability any better than $1/2$. So, if Definition 3.8 does not hold then $\mathcal{A}$ must (implicitly) be distinguishing the output of G from a

random string. The reduction we now show makes this explicit.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and define ε as

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] - \frac{1}{2}. \tag{3.2}$$

We use $\mathcal{A}$ to construct a distinguisher D for the pseudorandom generator G, such that D "succeeds" with probability $\varepsilon(n)$. The distinguisher is given a string w as input, and its goal is to determine whether w was chosen uniformly at random (i.e., w is a "random string") or whether w was generated by choosing a random k and computing $w := G(k)$ (i.e., w is a "pseudorandom string"). D emulates the eavesdropping experiment for $\mathcal{A}$ in the manner described below, and observes whether $\mathcal{A}$ succeeds or not. If $\mathcal{A}$ succeeds then D guesses that w must be a pseudorandom string, while if $\mathcal{A}$ does not succeed then D guesses that w is a random string. In detail:

> **Distinguisher D:**
> D is given as input a string $w \in \{0,1\}^{\ell(n)}$. (We assume that n can be determined from $\ell(n)$.)
>
> 1. Run $\mathcal{A}(1^n)$ to obtain a pair of messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.
> 2. Choose a random bit $b \leftarrow \{0,1\}$. Set $c := w \oplus m_b$.
> 3. Give c to $\mathcal{A}$ and obtain output b'. Output 1 if $b' = b$, and output 0 otherwise.

Before analyzing the behavior of D, we define a modified encryption scheme $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ that is exactly the one-time pad encryption scheme, except that we now incorporate a security parameter that determines the length of the messages to be encrypted. That is, $\widetilde{\mathsf{Gen}}(1^n)$ outputs a completely random key k of length $\ell(n)$, and the encryption of a message $m \in \ell(n)$ using the key $k \in \{0,1\}^{\ell(n)}$ is the ciphertext $c := k \oplus m$. (Decryption can be performed as usual, but is inessential to what follows.) By the perfect secrecy of the one-time pad, we have that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] = \frac{1}{2}. \tag{3.3}$$

In analyzing the behavior of D, the main observations are as follows:

1. If w is chosen uniformly at random from $\{0,1\}^{\ell(n)}$, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$. This is because $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w \in \{0,1\}^{\ell(n)}$ is a completely random string. It therefore follows that for $w \leftarrow \{0,1\}^{\ell(n)}$ chosen uniformly at random,

$$\Pr[D(w) = 1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] = \frac{1}{2},$$

where the second equality follows from Equation (3.3).

2. If w is equal to $G(k)$ for $k \leftarrow \{0,1\}^n$ chosen uniformly at random, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. This is because $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w = G(k)$ for a uniformly-distributed value $k \leftarrow \{0,1\}^n$. Thus, when $w = G(k)$ for $k \leftarrow \{0,1\}^n$ chosen uniformly at random, we have

$$\Pr[D(w) = 1] = \Pr[D(G(k)) = 1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] = \frac{1}{2} + \varepsilon(n)$$

where the second equality follows from the definition of ε.

Therefore,

$$\left|\Pr[D(w) = 1] - \Pr[D(G(k)) = 1]\right| = \varepsilon(n)$$

where, above, w is chosen uniformly from $\{0,1\}^{\ell(n)}$ and k is chosen uniformly from $\{0,1\}^n$. By the assumption that G is a pseudorandom generator, it must be the case that ε is negligible. By the definition of ε (see Equation (3.2)), this implies that Π has indistinguishable encryptions in the presence of an eavesdropper, concluding the proof. ■

It is easy to get lost in the details of the proof and wonder whether anything has been gained as compared to the one-time pad; after all, the one-time pad also encrypts an ℓ-bit message by XORing it with an ℓ-bit string! The point of the construction, of course, is that the ℓ-bit string $G(k)$ can be *much longer* than the key k. In particular, using the above encryption scheme it is possible to encrypt a file that is megabytes long using only a 128-bit key. This is in stark contrast with Theorem 2.7 that states that for any *perfectly*-secret encryption scheme, the key must be at least as long as the message being encrypted. The computational approach enables us to achieve much more than when perfect secrecy is required.

Reductions — a discussion. We do *not* prove unconditionally that Construction 3.15 is secure. Rather, we prove that it is secure *under the assumption* that G is a pseudorandom generator. This approach of *reducing* the security of a construction to some underlying primitive is of great importance for a number of reasons. First, as we have already noted, we do not know how to prove the unconditional existence of an encryption scheme satisfying Definition 3.8 and such a proof seems far out of reach today. Given this, reducing the security of a higher-level construction to a lower-level primitive has a number of advantages (as discussed in Section 1.4.2). One of these advantages is the fact that, in general, it is easier to design a lower-level primitive than a higher-level one; it is similarly easier, in general, to be convinced that something satisfies a lower-level definition than a higher-level one. This does not mean that constructing a pseudorandom generator is "easy", only that it is easier than constructing an encryption scheme from scratch. (Of course, in the present case the encryption scheme does almost nothing except XOR the

output of a pseudorandom generator with the message and so this isn't really true. However, we will see more complex constructions and in these cases the ability to reduce the task to a simpler one is of great importance.)

3.4.2 Handling Variable-Length Messages

The construction of the previous section has the disadvantage of allowing encryption only of *fixed-length* messages. (I.e., for each particular value n of the security parameter, only messages of length $\ell(n)$ can be encrypted.) This deficiency is easy to address by using a *variable output-length pseudorandom generator* in Construction 3.15.

Variable output-length pseudorandom generators. In some applications, we do not know ahead of time how many bits of pseudorandomness will be needed. Thus, what we actually want is a pseudorandom generator that can output a pseudorandom string of any desired length. More specifically, we would like G to receive two inputs: the seed s and the length of the output ℓ (the length of ℓ is given in unary for the same reason that the security parameter is given in unary); G should then output a pseudorandom string of length ℓ. We now present the formal definition:

DEFINITION 3.17 *A deterministic polynomial-time algorithm G is a* variable output-length pseudorandom generator *if the following hold:*

1. *Let s be a string and $\ell > 0$ be an integer. Then $G(s, 1^\ell)$ outputs a string of length ℓ.*

2. *For all s, ℓ, ℓ' with $\ell < \ell'$, the string $G(s, 1^\ell)$ is a prefix of $G(s, 1^{\ell'})$.*[7]

3. *Define $G_\ell(s) \stackrel{\text{def}}{=} G(s, 1^{\ell(|s|)})$. Then for every polynomial $\ell(\cdot)$ it holds that G_ℓ is a pseudorandom generator with expansion factor ℓ.*

Any standard pseudorandom generator (as in Definition 3.14) can be converted into a variable output-length one; see Section 6.4.2.

Given the above definition, we modify Construction 3.15 in the natural way: encryption of a message m using the key k is carried out by computing the ciphertext $c := G(k, 1^{|m|}) \oplus m$; decryption of a ciphertext c using the key k is carried out by computing the message $m := G(k, 1^{|c|}) \oplus c$. We leave it as an exercise to prove that this scheme also satisfies Definition 3.8. The proof follows that of Theorem 3.16 except for one technical subtlety that arises.

[7] This condition is needed for technical reasons in order to prove the security of Construction 3.15 for variable-length messages when using a variable-length generator.

3.4.3 Stream Ciphers and Multiple Encryptions

In the cryptographic literature, an encryption scheme of the type presented in the previous two sections is often called a stream cipher. This is due to the fact that encryption is carried out by first generating a *stream* of pseudorandom bits, and then XORing this stream with the plaintext. Unfortunately, there is a bit of confusion as to whether the term "stream cipher" refers to the *algorithm* that generates the stream (i.e., the pseudorandom generator G) or to the entire encryption scheme. This is a crucial issue because *the way a pseudorandom generator is used determines whether or not a given encryption scheme is secure.* In our opinion, it is best to use the term stream cipher to refer to the algorithm that generates the pseudorandom stream, and thus a "secure" stream cipher should satisfy the definition of a variable output-length pseudorandom generator.[8] Using this terminology, a stream cipher is not an encryption scheme per se, but rather a *tool* for constructing encryption schemes. The importance of this discussion will become clearer when we discuss the issue of multiple encryptions below.

Stream ciphers in practice. There are a number of practical constructions of stream ciphers available, and these are typically extraordinarily fast. A popular example is the stream cipher RC4 which is widely considered to be secure when used appropriately (see below). The security of practical stream ciphers is not yet very well understood, particularly in comparison to *block ciphers* (introduced later in this chapter). This is borne out by the fact that there is no standardized, popular stream cipher that has been used for many years and whose security has not come into question. For example, "plain" RC4 (that was considered secure at one point and is still widely deployed) is now known to have some significant weaknesses. For one, the first few bytes of the output stream generated by RC4 have been shown to be biased. Although this may seem benign, it was also shown that this weakness can be used to feasibly break the WEP encryption protocol used in 802.11 wireless networks. (WEP is a standardized protocol for protecting wireless communications. The WEP standard has since been updated to fix the problem.) If RC4 is to be used, the first 1024 bits or so of the output stream should be discarded.

Linear feedback shift registers (LFSRs) have, historically, also been popular as stream ciphers. However, they have been shown to be horribly insecure (to the extent that the key can be completely recovered given sufficiently-many bytes of the output) and so should never be used today.

In general, we advocate the use of block ciphers in constructing secure encryption schemes. Block ciphers are efficient enough for all but the most resource-constrained environments, and seem to be more secure than existing stream ciphers. For completeness, we remark that a stream cipher can be

[8] Soon we will introduce the notion of a *block cipher*. In that context, it is accepted that this term refers to the tool itself and not how it is used in order to achieve secure encryption. We therefore prefer to use the term "stream cipher" analogously.

easily constructed from a block cipher, as described in Section 3.6.4 below. The disadvantage of this approach as compared to a dedicated stream cipher is that it is usually less efficient.

Security for Multiple Encryptions

Definition 3.8, and all our discussion until now, has dealt with the case that the adversary receives a *single* ciphertext. In reality, however, communicating parties send multiple ciphertexts to each other and an eavesdropper will see many of these. It is therefore of great importance to ensure that the encryption scheme being used is secure even in this setting.

Let us first give a definition of security. As in the case of Definition 3.8, we first introduce an appropriate experiment that is defined for an encryption scheme Π, an adversary $\mathcal{A}$, and a security parameter n:

The multiple-message eavesdropping experiment $\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$:

1. *The adversary $\mathcal{A}$ is given input 1^n, and outputs a pair of* vectors *of messages $\vec{M}_0 = (m_0^1, \ldots, m_0^t)$ and $\vec{M}_1 = (m_1^1, \ldots, m_1^t)$ with $|m_0^i| = |m_1^i|$ for all i.*
2. *A key k is generated by running $\mathsf{Gen}(1^n)$, and a random bit $b \leftarrow \{0,1\}$ is chosen. For all i, the ciphertext $c^i \leftarrow \mathsf{Enc}_k(m_b^i)$ is computed and the vector of ciphertexts $\vec{C} = (c^1, \ldots, c^t)$ is given to $\mathcal{A}$.*
3. *$\mathcal{A}$ outputs a bit b'.*
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.*

The definition of security for multiple messages is the same as for a single message, except that it now refers to the above experiment. That is:

DEFINITION 3.18 *A private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ has* indistinguishable multiple encryptions in the presence of an eavesdropper *if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{PrivK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over the random coins used by $\mathcal{A}$, as well as the random coins used in the experiment (for choosing the key and the random bit b, as well as for the encryption itself).

A crucial observation is that security for a *single* encryption (as in Definition 3.8) does not imply security under *multiple* encryptions. This is formalized in the following proposition.

PROPOSITION 3.19 *There exist private-key encryption schemes that have indistinguishable encryptions in the presence of an eavesdropper but do not have indistinguishable multiple encryptions in the presence of an eavesdropper.*

PROOF We do not have to look far to find an encryption scheme fulfilling the proposition. Specifically, Construction 3.15, that was proven secure for a single encryption in Theorem 3.16, is not secure when used for multiple encryptions. This should not come as a surprise because we have already seen that the one-time pad is only secure when used once, and Construction 3.15 works in a similar way.

Concretely, consider the following adversary $\mathcal{A}$ attacking the encryption scheme (in the sense defined by experiment $\mathsf{PrivK}^{\mathsf{mult}}$): $\mathcal{A}$ outputs the vectors $\vec{M}_0 = (0^n, 0^n)$ and $\vec{M}_1 = (0^n, 1^n)$. That is, the first vector contains two plaintexts, where each plaintext is just a length-n string of zeroes. In contrast, in the second vector the first plaintext is all zeroes and the second is all ones. Now, let $\vec{C} = (c^1, c^2)$ be the vector of ciphertexts that $\mathcal{A}$ receives. If $c^1 = c^2$, then $\mathcal{A}$ outputs 0; otherwise, $\mathcal{A}$ outputs 1.

We now analyze $\mathcal{A}$'s success in guessing b. The main point is that Construction 3.15 is *deterministic*, so that if the same message is encrypted multiple times then the same ciphertext results each time. Now, if $b = 0$ then the same message is encrypted each time (since $m_0^1 = m_0^2$); thus, $c^1 = c^2$ and hence $\mathcal{A}$ always outputs 0 in this case. On the other hand, if $b = 1$ then a different message is encrypted each time (since $m_1^1 \neq m_1^2$) and $c^1 \neq c^2$; here, $\mathcal{A}$ always outputs 1. We conclude that $\mathcal{A}$ outputs $b' = b$ with probability 1 and so the encryption scheme is not secure with respect to Definition 3.18. ■

The proof of Proposition 3.19 may seem contrived. However, this is far from the truth. The mere knowledge that the same message has been re-sent can provide significant information and has historically been very useful to cryptanalysts.

Necessity of probabilistic encryption. In the proof of Proposition 3.19 we have shown that Construction 3.15 is not secure for multiple encryptions. The only feature of that construction used in the proof was that encrypting a message always yields the same ciphertext, and so we actually obtain that *any* deterministic scheme must be insecure for multiple encryptions. This is important enough to state as a theorem.

THEOREM 3.20 *Let* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be an encryption scheme for which* Enc *is a deterministic function of the key and the message. Then* Π *does not have indistinguishable multiple encryptions in the presence of an eavesdropper.*

This implies that to construct an encryption scheme that is secure with respect to Definition 3.18, at a very minimum we will have to ensure that when the same message is encrypted multiple times, a different ciphertext results each time. At first sight this may seem like an impossible task since decryption must always be able to recover the message. However, we will later see how to achieve it.

Multiple encryptions using a stream cipher — a common error. Unfortunately, incorrect implementations of cryptographic constructions are very frequent. One common error is to use a stream cipher (in its naive form as in Construction 3.15) in order to encrypt multiple plaintexts. As an example, this error appears in an implementation of encryption in Microsoft Word and Excel; see [147]. In practice, such an error can be devastating. We emphasize that this is not just a "theoretical artifact" due to the fact that encrypting the same message twice yields the same message. Even if the same message is never encrypted twice, various attacks are possible, as mentioned in Section 2.2.

Secure Multiple Encryptions Using a Stream Cipher

There are typically two ways in which a stream cipher/pseudorandom generator is used in practice to securely encrypt multiple plaintexts (see also Figure 3.3):

1. *Synchronized mode:* In this mode, the communicating parties use a different part of the stream output by the stream cipher in order to encrypt each message. This mode is "synchronized" because both parties need to know which parts of the stream have already been used in order to prevent re-use, which (as we have already shown) is not secure.

   This mode is useful in a setting where parties are communicating in a single session. In this setting, the first party uses the first part of the stream in order to send its first message. The second party obtains the ciphertext, decrypts, and then uses the next part of the stream in order to encrypt its reply. Since each part of the stream is used only once, it is possible to view the concatenation of all of the messages sent by the parties as a *single* long plaintext. Security of the scheme follows as in Theorem 3.16 (adapted for the case of variable-length messages).

   This mode is not suitable in all applications because the parties are required to maintain state between encryptions (in particular, to tell them which portion of the stream to use next). For the same reason, the security of this method does not contradict Theorem 3.20 even though it is deterministic (because it does not satisfy the syntactic requirements of Definition 3.7).

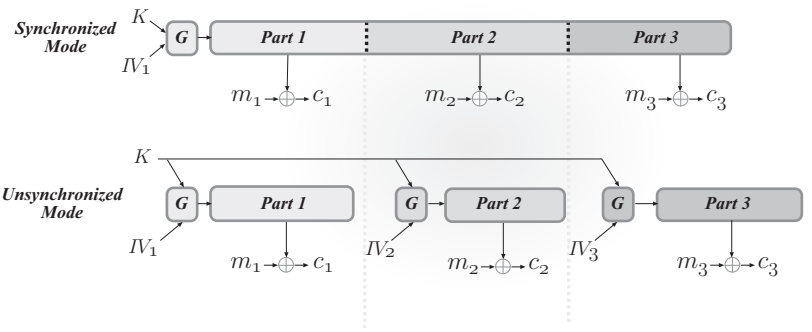

FIGURE 3.3: Synchronized mode vs. unsynchronized mode.

2. *Unsynchronized mode:* In this mode, encryptions are carried out independently of one another and the parties do not need to maintain state. In order to achieve security, however, our notion of a pseudorandom generator must be significantly strengthened. Now, we view a pseudorandom generator as taking two inputs: a *seed* s and an *initial vector* IV of length n. Roughly speaking, the requirement is that $G(s, IV)$ is pseudorandom even when IV is known (but s is kept secret). Furthermore, for two randomly-chosen initial vectors IV_1 and IV_2, the streams $G(s, IV_1)$ and $G(s, IV_2)$ should remain pseudorandom even when viewed together and with their respective IVs.

   Given a generator as above, encryption can be defined as

$$\mathsf{Enc}_k(m) := \langle IV,\, G(k, IV) \oplus m \rangle$$

   where $IV \leftarrow \{0,1\}^n$ is chosen at random. (For simplicity, we focus on encrypting fixed-length messages.) The IV is chosen fresh (i.e., independently and uniformly at random) for each encryption and thus each stream is pseudorandom, even if previous streams are known. Note that the IV is sent as part of the ciphertext in order to enable the recipient to decrypt; i.e., given (IV, c), the recipient can compute $m := c \oplus G(k, IV)$.

   Many stream ciphers in practice are assumed to have this *augmented pseudorandomness* property sketched informally above, and can thus be used in unsynchronized mode. However, we warn that a standard pseudorandom generator may not have this property, and that this assumption is a strong one. In fact, such a generator is "almost" a pseudorandom function; see Section 3.6.1 and Exercise 3.20.

3.5 Security Against Chosen-Plaintext Attacks (CPA)

Until now we have considered a relatively weak adversary who only passively eavesdrops on the communication between two honest parties. (Of course, our actual definition of $\mathsf{PrivK}^{\mathsf{eav}}$ allows the adversary to choose the plaintexts that are to be encrypted. Nevertheless, beyond this capability the adversary is completely passive.) In this section, we formally introduce a more powerful type of adversarial attack, called a chosen-plaintext attack (CPA). The definition of security under CPA attacks is the same as in Definition 3.8, except that the adversary's attack capabilities are strengthened.

The basic idea behind a chosen-plaintext attack is that the adversary $\mathcal{A}$ is allowed to ask for encryptions of multiple messages chosen adaptively. This is formalized by allowing $\mathcal{A}$ to interact freely with an encryption oracle, viewed as a "black-box" that encrypts messages of $\mathcal{A}$'s choice using the secret key k (that is unknown to $\mathcal{A}$). Following standard notation in computer science, we denote by $\mathcal{A}^{\mathcal{O}(\cdot)}$ the computation of $\mathcal{A}$ given access to an oracle $\mathcal{O}$, and thus in this case we denote the computation of $\mathcal{A}$ with access to an encryption oracle that uses key k by $\mathcal{A}^{\mathsf{Enc}_k(\cdot)}$. When $\mathcal{A}$ *queries* its oracle by providing it with a plaintext message m as input, the oracle returns a ciphertext $c \leftarrow \mathsf{Enc}_k(m)$ as the reply. When Enc is randomized, the oracle uses fresh random coins each time it answers a query.

The definition of security requires that $\mathcal{A}$ should not be able to distinguish the encryption of two arbitrary messages, *even when $\mathcal{A}$ is given access to an encryption oracle*. We present the definition and afterwards discuss what real-world adversarial attacks the definition is meant to model.

We first define an experiment for any private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, any adversary $\mathcal{A}$, and any value n of the security parameter:

The CPA indistinguishability experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$:

1. *A key k is generated by running $\mathsf{Gen}(1^n)$.*
2. *The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\mathsf{Enc}_k(\cdot)$, and outputs a pair of messages m_0, m_1 of the same length.*
3. *A random bit $b \leftarrow \{0,1\}$ is chosen, and then a ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$. We call c the* challenge ciphertext.
4. *The adversary $\mathcal{A}$ continues to have oracle access to $\mathsf{Enc}_k(\cdot)$, and outputs a bit b'.*
5. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. (In case $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1$, we say that $\mathcal{A}$* succeeded.)

DEFINITION 3.21 *A private-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable encryptions under a chosen-plaintext attack (*or is* CPA-secure) *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over the random coins used by $\mathcal{A}$*, as well as the random coins used in the experiment.*

Any scheme that has indistinguishable encryptions under a chosen-plaintext attack clearly also has indistinguishable encryptions in the presence of an eavesdropper. This holds because $\mathsf{PrivK}^{\mathsf{eav}}$ is a special case of $\mathsf{PrivK}^{\mathsf{cpa}}$ where the adversary doesn't use its oracle at all.

At first sight, it may appear that Definition 3.21 is impossible to achieve. In particular, consider an adversary that outputs (m_0, m_1) and then receives the challenge ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$. Since the adversary $\mathcal{A}$ has oracle access to $\mathsf{Enc}_k(\cdot)$, it can request that this oracle encrypt the messages m_0 and m_1 and thus obtain $c_0 \leftarrow \mathsf{Enc}_k(m_0)$ and $c_1 \leftarrow \mathsf{Enc}_k(m_1)$. Adversary $\mathcal{A}$ can then compare c_0 and c_1 to c: if $c = c_0$ then, seemingly, $\mathcal{A}$ knows that $b = 0$, and if $c = c_1$ then it knows that $b = 1$. Why doesn't this strategy allow $\mathcal{A}$ to determine b with probability 1?

The answer is that such an attack would indeed work if the encryption scheme was deterministic (because this implies that every time a message is encrypted, the same ciphertext is obtained). Thus, as with security under multiple encryptions, no *deterministic* encryption scheme can be secure against chosen-plaintext attacks. Rather, any CPA-secure encryption scheme *must* be probabilistic. That is, it must use random coins as part of the encryption process in order to ensure that two encryptions of the same message are likely to be different.[9]

Chosen-plaintext attacks in the real world. Definition 3.21 is at least as strong as our earlier Definition 3.8, and so certainly no security is lost by working with this newer definition. In general, however, there may be a price for using a definition that is too strong because it may cause us to use less efficient schemes (even though there are more efficient ones that would suffice for "real-world applications"). We should therefore ask ourselves whether chosen-plaintext attacks represent a realistic adversarial threat with which we should really be concerned.

The fact is that chosen-plaintext attacks (in one form or another) are a realistic threat in many scenarios. We demonstrate this by first looking at

[9] As we have seen, if the encryption process maintains state between successive encryptions (as in the synchronized mode for stream ciphers), random coin tosses may not be necessary. As per Definition 3.7, we typically consider only stateless schemes (which are preferable).

some military history from World War II. In May 1942, US Navy cryptanalysts had discovered that Japan was planning an attack on Midway island in the Central Pacific. They had learned this by intercepting a communication message containing the ciphertext fragment "AF" that they believed corresponded to the plaintext "Midway island". Unfortunately, their attempts to convince Washington planners that this was indeed the case were futile; the general belief was that Midway could not possibly be the target. The Navy cryptanalysts then devised the following plan. They instructed the US forces at Midway to send a plaintext message that their freshwater supplies were low. The Japanese intercepted this message and immediately reported to their superiors that "AF" was low on water. The Navy cryptanalysts now had their proof that "AF" was indeed Midway, and the US forces dispatched three aircraft carriers to the location. The result is that Midway was saved, and the Japanese incurred great losses. It is even said that this battle was the turning point in the war by the US against Japan in the Pacific. (See [91, 146] for more information.)

The Navy cryptanalysts here carried out a classic chosen-plaintext attack. They were able to request the Japanese (albeit in a roundabout way) to encrypt the word "Midway" in order to learn information about another ciphertext that they had previously intercepted. If the Japanese encryption scheme had been secure against chosen-plaintext attacks, this strategy by the US cryptanalysts would not have worked (and history may have turned out very differently)! We stress that the Japanese never intended to act as an "encryption oracle" for the US and thus were the Japanese to analyze the necessity for CPA security, it is unlikely they would have concluded that it was necessary.[10]

We warn against thinking that chosen-plaintext attacks are only the result of clever manipulation. There are many cases where an adversary can influence what is encrypted by an honest party (even if it is more unusual for the adversary to be in complete control over what is encrypted). Consider the following example: many servers communicate with each other today in a secured way (i.e., using encryption). However, the messages that these servers send to each other are based on internal and external requests that they receive, which are in turn chosen by users that may actually be malicious. These attackers can influence the plaintext messages that the servers encrypt, sometimes to a great extent. Such systems must be protected by using an encryption scheme that is secure against chosen-plaintext attacks, and we therefore strongly encourage always using an encryption scheme of this sort.

CPA security for multiple encryptions. The extension of Definition 3.21 to the case of multiple encryptions is straightforward and is the same as the extension of Definition 3.8 to Definition 3.18. That is, we define an experiment

[10] Ask yourself whether you would have anticipated such an attack.

which is exactly the same as $\mathsf{PrivK}^{\mathsf{cpa}}$ except that $\mathcal{A}$ outputs a pair of *vectors* of plaintexts. Then, we require that no polynomial-time $\mathcal{A}$ can succeed in the experiment with probability that is non-negligibly greater than 1/2.

Importantly, CPA security for a single encryption *automatically implies* CPA security for multiple encryptions. (This stands in contrast to the case of eavesdropping adversaries; see Proposition 3.19.) We state the claim here without proof; a similar claim, but in the public-key setting, is proved in Section 10.2.2.

PROPOSITION 3.22 *Any private-key encryption scheme that has indistinguishable encryptions under a chosen-plaintext attack also has indistinguishable* multiple *encryptions under a chosen-plaintext attack.*

This is a significant technical advantage of CPA security. It suffices to prove that a scheme is CPA-secure for a single encryption and then we obtain "for free" that it is CPA-secure for multiple encryptions as well.

Fixed-length vs. arbitrary-length messages. Another advantage of working with the definition of CPA-security is that it allows us to treat fixed-length encryption schemes without much loss of generality. In particular, we claim that given any CPA-secure *fixed-length* encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, it is possible to construct a CPA-secure encryption scheme $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ for *arbitrary-length* messages quite easily. For simplicity, say Π encrypts messages that are 1-bit long (though everything we say extends in the natural way when Π encrypts messages of some arbitrary length $\ell(n)$). Leave Gen' the same as Gen. Define Enc'_k for any message m (having some arbitrary length ℓ) in the following way:

$$\mathsf{Enc}'_k(m) = \mathsf{Enc}_k(m_1), \ldots, \mathsf{Enc}_k(m_\ell),$$

where $m = m_1 \cdots m_\ell$ and $m_i \in \{0,1\}$ for all i. Decryption is done in the natural way. We claim that Π' is CPA-secure if and only if Π is. A proof follows from Proposition 3.22.

Notwithstanding the above, there may in practice be more efficient ways to encrypt messages of arbitrary length than by adapting a fixed-length encryption scheme in the above manner. We treat other ways of encrypting arbitrary-length messages in Section 3.6.4.

3.6 Constructing CPA-Secure Encryption Schemes

In this section we will construct encryption schemes that are secure against chosen-plaintext attacks. We begin by introducing the important notion of *pseudorandom functions*.

3.6.1 Pseudorandom Functions

As we have seen, pseudorandom generators can be used to obtain security in the presence of eavesdropping adversaries. The notion of pseudorandomness is also instrumental in obtaining security against chosen-plaintext attacks. Now, however, instead of considering pseudorandom *strings*, we consider pseudorandom *functions*. We will specifically be interested in pseudorandom functions mapping n-bit strings to n-bit strings. As in our earlier discussion of pseudorandomness, it does not make much sense to say that any *fixed* function $f : \{0,1\}^n \to \{0,1\}^n$ is pseudorandom (in the same way that it makes little sense to say that any fixed function is random). Thus, we must technically refer to the pseudorandomness of a *distribution* on functions. An easy way to do this is to consider *keyed functions*, defined next.

A keyed function F is a two-input function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, where the first input is called the key and denoted k, and the second input is just called the input. In general the key k will be chosen and then *fixed*, and we will then be interested in the single-input function $F_k : \{0,1\}^* \to \{0,1\}^*$ defined by $F_k(x) \stackrel{\text{def}}{=} F(k,x)$. For simplicity, we will generally assume that F is length-preserving meaning that the key, input, and output lengths of F are all the same. That is, we assume that the function F is only defined when the key k and the input x have the same length, in which case $|F_k(x)| = |x| = |k|$. So, by fixing a key $k \in \{0,1\}^n$ we obtain a function $F_k(\cdot)$ mapping n-bit strings to n-bit strings. We say that F is efficient if there is a deterministic polynomial-time algorithm that computes $F(k,x)$ given k and x as input. We will only be interested in functions F that are efficient.

A keyed function F induces a natural distribution on functions given by choosing a random key $k \leftarrow \{0,1\}^n$ and then considering the resulting single-input function F_k. Intuitively, we call F *pseudorandom* if the function F_k (for a randomly-chosen key k) is indistinguishable from a function chosen uniformly at random from the set of all functions having the same domain and range; that is, if no polynomial-time adversary can distinguish — in a sense we will more carefully define soon — whether it is interacting with F_k (for randomly-chosen key k) or f (where f is chosen at random from the set of all functions mapping n-bit strings to n-bit strings).

Since the notion of choosing a function at random is less familiar than the notion of choosing a string at random, it is worth spending a bit more time on this idea. From a mathematical point of view, we can consider the set Func_n of all functions mapping n-bit strings to n-bit strings. This set is finite (as we will see in a moment), and so randomly selecting a function mapping n-bit strings to n-bit strings corresponds exactly to choosing an element uniformly at random from this set. How large is the set Func_n? A function f is fully specified by giving its value on each point in its domain. In fact, we can view any function (over a finite domain) as a large look-up table that stores $f(x)$ in the row of the table labeled by x. For $f \in \mathsf{Func}_n$, the look-up table for f has 2^n rows (one for each point of the domain $\{0,1\}^n$) and each row contains

an n-bit string (since the range of f is $\{0,1\}^n$). Any such table can thus be represented using exactly $n \cdot 2^n$ bits. Moreover, the functions in Func_n are in one-to-one correspondence with look-up tables of this form, meaning that they are in one-to-one correspondence with all strings of length $n \cdot 2^n$. Since there are $2^{n \cdot 2^n}$ strings of length $n \cdot 2^n$, this is the exact size of Func_n.

Viewing a function as a look-up table provides another useful way to think about selecting a function $f \in \mathsf{Func}_n$ uniformly at random: It is exactly equivalent to choosing each row of the look-up table of f uniformly at random. That is, the values $f(x)$ and $f(y)$ (for $x \neq y$) are completely independent and uniformly distributed.

Coming back to our discussion of pseudorandom functions, recall that we wish to construct a keyed function F such that F_k (for $k \leftarrow \{0,1\}^n$ chosen uniformly at randomly) is indistinguishable from f (for $f \leftarrow \mathsf{Func}_n$ chosen uniformly at random). The former is chosen from a distribution over (at most) 2^n distinct functions, whereas the latter is chosen from a distribution over all $2^{n \cdot 2^n}$ functions in Func_n. Despite this, the "behavior" of these functions must look the same to any polynomial-time distinguisher.

A first attempt at formalizing the notion of a pseudorandom function would be to proceed in the same way as in Definition 3.14. That is, we could require that every polynomial-time distinguisher D that receives a description of the pseudorandom function F_k outputs 1 with "almost" the same probability as when it receives a description of a random function f. However, this definition is inappropriate since the description of a random function has *exponential length* (i.e., given by its look-up table which has length $n \cdot 2^n$), while D is limited to running in polynomial time. So, D would not even have sufficient time to examine its entire input.

The actual definition therefore gives D *oracle access* to the function in question (either F_k or f). D is allowed to query the oracle at any point x, in response to which the oracle returns the value of the function evaluated at x. We treat this oracle as a black-box in the same way as when we provided the adversary with oracle access to the encryption procedure in the definition of a chosen-plaintext attack. Here, however, the oracle computes a deterministic function, and so always returns the same result when queried twice on the same input. We are now ready to present the formal definition. (Although the definition requires that F be length-preserving, this is merely a simplifying assumption that is in no way necessary.)

DEFINITION 3.23 *Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. We say that F is a* pseudorandom function *if for all probabilistic polynomial-time distinguishers D, there exists a negligible function* negl *such that:*

$$\left| \Pr[D^{F_k(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1] \right| \leq \mathsf{negl}(n),$$

where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random and f is chosen uniformly at random from the set of functions mapping n-bit strings to n-bit strings.

Notice that D interacts freely with its oracle. Thus, it can ask queries adaptively, choosing the next input based on the previous outputs received. However, since D runs in polynomial time, it can ask only a polynomial number of queries. Notice also that a pseudorandom function must inherit any efficiently checkable property of a random function. For example, even if x and x' differ in only a single bit, the outputs $F_k(x)$ and $F_k(x')$ must (with overwhelming probability over choice of k) look completely uncorrelated. This gives a hint as to why pseudorandom functions are useful for constructing secure encryption schemes.

An important point in the definition is that the distinguisher D is *not* given the key k. It is meaningless to require that F_k be pseudorandom if k is known, since given k it is trivial to distinguish an oracle for F_k from an oracle for f. All the distinguisher has to do is query the oracle at the point 0^n to obtain the answer y, and compare this to the result $y' = F_k(0^n)$ that can be computed using the known value k. An oracle for F_k will always return $y = y'$, while an oracle for a random function will have $y = y'$ with probability only 2^{-n}. In practice, this means that once k is revealed, all claims to the pseudorandomness of F_k no longer hold. To take a concrete example, consider a pseudorandom function F. Then given oracle access to F_k (for random k), it must be hard to find an input x for which $F_k(x) = 0^n$ (since it would be hard to find such an input for a truly random function f). But if k is known, then finding such an input may be easy (and in reality often is).

On the existence of pseudorandom functions. As with pseudorandom generators, it is important to ask whether pseudorandom functions exist and, if so, under what assumptions. In practice, very efficient primitives called *block ciphers* are used and are widely believed to act as pseudorandom functions. This is discussed further in Section 3.6.3, and a more in-depth treatment of block ciphers appears in Chapter 5. From a theoretical point of view, it is known that pseudorandom functions exist if and only if pseudorandom generators exist, and so pseudorandom functions can be constructed based on any of the hard problems that allow the construction of pseudorandom generators. This is discussed at length in Chapter 6. The existence of pseudorandom functions based on these hard problems represents one of the surprising and truly amazing contributions of modern cryptography.

Using pseudorandom functions in cryptography. Pseudorandom functions turn out to be a very useful building block for a number of different cryptographic constructions. We use them below to obtain CPA-secure encryption and in Chapter 4 to construct message authentication codes. One of the reasons that they are so useful is that they enable a clean and elegant analysis of the constructions that use them. That is, given a scheme that is based on a pseudorandom function, a general way of analyzing the scheme is to first prove its security under the assumption that a truly random function is used instead. This step relies on a probabilistic analysis and has nothing to do with computational bounds or hardness. Next, the security of the original

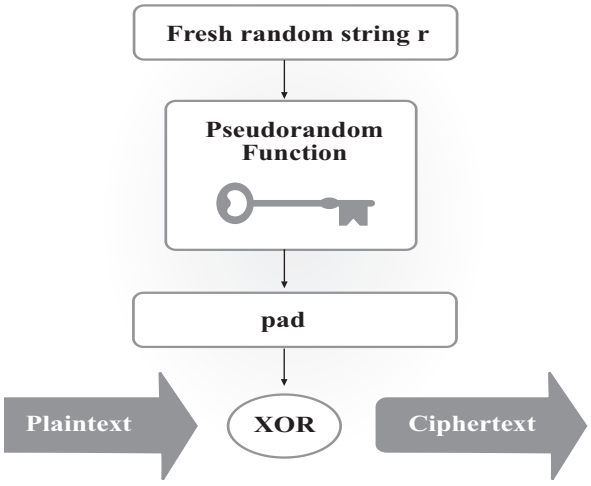

FIGURE 3.4: Encryption with a pseudorandom function.

scheme is derived by proving that if an adversary can break the scheme when a pseudorandom function is used, then it must implicitly be distinguishing the function from random.

3.6.2 CPA-Secure Encryption Schemes from Pseudorandom Functions

We focus here on constructing a fixed-length encryption scheme that is CPA-secure. By what we have said at the end of Section 3.5, this implies the existence of a CPA-secure encryption scheme for arbitrary-length messages. In Section 3.6.4 we will consider more efficient ways of handling messages of arbitrary length.

A naive attempt at constructing a secure encryption scheme from a pseudorandom function is to define $\mathsf{Enc}_k(m) = F_k(m)$. On the one hand, we expect that this "reveals no information about m" (since $f(m)$ for a *random* function f is simply a random n-bit string). However, this method of encryption is *deterministic* and so cannot possibly be CPA-secure. Concretely, given $c = \mathsf{Enc}_k(m_b)$ it is possible to request an encryption of $\mathsf{Enc}_k(m_0)$ and $\mathsf{Enc}_k(m_1)$; since $\mathsf{Enc}_k(\cdot) = F_k(\cdot)$ is a deterministic function, one of the encryptions will equal c and thus reveal the value of b.

Our actual construction is *probabilistic*. Specifically, we encrypt by applying the pseudorandom function to a *random value* r (rather than the plaintext message) and XORing the result with the plaintext. (See Construction 3.24 and Figure 3.4.) This can again be viewed as an instance of XORing a pseudorandom "pad" with a plaintext message, with the major difference being the fact that an *independent* pseudorandom pad is used each time. This holds as

long as the pseudorandom function is applied to a different input each time. Of course, it is possible that a random value r repeats and is used more than once and this will explicitly be taken into account in our proof.

CONSTRUCTION 3.24

Let F be a pseudorandom function. Define a private-key encryption scheme for messages of length n as follows:

- Gen: on input 1^n, choose $k \leftarrow \{0,1\}^n$ uniformly at random and output it as the key.
- Enc: on input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^n$, choose $r \leftarrow \{0,1\}^n$ uniformly at random and output the ciphertext

$$c := \langle r,\, F_k(r) \oplus m \rangle.$$

- Dec: on input a key $k \in \{0,1\}^n$ and a ciphertext $c = \langle r, s \rangle$, output the plaintext message

$$m := F_k(r) \oplus s.$$

A CPA-secure encryption scheme from any pseudorandom function.

Intuitively, security holds because $F_k(r)$ looks completely random to an adversary who observes a ciphertext $\langle r, s \rangle$ — and thus the encryption scheme is similar to the one-time pad — *as long as the value r was not used in some previous encryption* (specifically, as long as it was not used by the encryption oracle when answering one of the adversary's queries). Moreover, this "bad event" (namely, a repeating value of r) occurs with only negligible probability.

THEOREM 3.25 *If F is a pseudorandom function, then Construction 3.24 is a fixed-length private-key encryption scheme for messages of length n that has indistinguishable encryptions under a chosen-plaintext attack.*

PROOF The proof here follows a general paradigm for working with pseudorandom functions: First, we analyze the security of the scheme in an idealized world where a truly random function f is used in place of F_k, and show that the scheme is secure in this case. Next, we claim that if the scheme were insecure when F_k was used then this would imply the possibility of distinguishing F_k from a truly random function.

Let $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ be an encryption scheme that is exactly the same as $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ in Construction 3.24, except that a truly random function f is used in place of F_k. That is, $\widetilde{\mathsf{Gen}}(1^n)$ chooses a random function $f \leftarrow \mathsf{Func}_n$, and $\widetilde{\mathsf{Enc}}$ encrypts just like Enc except that f is used instead of F_k. (This is not a legal encryption scheme because it is not efficient. Nevertheless,

this is a mental experiment for the sake of the proof, and is well defined for this purpose.) We claim that for every (even inefficient) adversary $\mathcal{A}$ that makes at most $q(n)$ queries to its encryption oracle, we have

$$\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] \leq \frac{1}{2} + \frac{q(n)}{2^n}. \tag{3.4}$$

To see this, recall that every time a message m is encrypted (either by the encryption oracle or when the challenge ciphertext in experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)$ is computed), a random $r \leftarrow \{0,1\}^n$ is chosen and the ciphertext is set equal to $\langle r, f(r) \oplus m\rangle$. Let r_c denote the random string used when generating the challenge ciphertext $c = \langle r_c, f(r_c) \oplus m_b\rangle$. There are two subcases:

1. *The value r_c is used by the encryption oracle to answer at least one of $\mathcal{A}$'s queries:* In this case, $\mathcal{A}$ may easily determine which of its messages was encrypted. This is so because whenever the encryption oracle returns a ciphertext $\langle r, s\rangle$ in response to a request to encrypt the message m, the adversary learns the value of $f(r)$ (since $f(r) = s \oplus m$).

   Since $\mathcal{A}$ makes at most $q(n)$ queries to its oracle and each oracle query is answered using a value r chosen uniformly at random, the probability of this event is at most $q(n)/2^n$ (as can be seen by applying a union bound).

2. *The value r_c is never used by the encryption oracle to answer any of $\mathcal{A}$'s queries:* In this case, $\mathcal{A}$ learns nothing about the value of $f(r_c)$ from its interaction with the encryption oracle (since f is a truly random function). That means that, as far as $\mathcal{A}$ is concerned, the value $f(r_c)$ that is XORed with m_b is completely random, and so the probability that $\mathcal{A}$ outputs $b' = b$ in this case is exactly $1/2$ (as in the case of the one-time pad).

Let Repeat denote the event that r_c is used by the encryption oracle to answer at least one of $\mathcal{A}$'s queries. We have shown above that the probability that Repeat occurs is at most $q(n)/2^n$, and that the probability that $\mathcal{A}$ succeeds in $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}$ if Repeat does not occur is exactly $1/2$. Thus, we have:

$$\begin{aligned}
&\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \\
&\quad = \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1 \wedge \mathsf{Repeat}] + \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1 \wedge \overline{\mathsf{Repeat}}] \\
&\quad \leq \Pr[\mathsf{Repeat}] + \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1 \mid \overline{\mathsf{Repeat}}] \\
&\quad \leq \frac{q(n)}{2^n} + \frac{1}{2},
\end{aligned}$$

as stated in Equation (3.4).

Now, fix some PPT adversary $\mathcal{A}$ and define the function ε by

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right] - \frac{1}{2}. \tag{3.5}$$

The number of oracle queries made by $\mathcal{A}$ is upper bounded by its running-time. Since $\mathcal{A}$ runs in polynomial-time, the number of oracle queries it makes is upper bounded by some polynomial $q(\cdot)$. Equation (3.4) also holds with respect to this $\mathcal{A}$; thus:

$$\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \leq \frac{1}{2} + \frac{q(n)}{2^n}$$

and by the definition of ε,

$$\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1] = \frac{1}{2} + \varepsilon(n).$$

If ε is *not* negligible, then the difference between these is also not negligible. Intuitively, such a "gap" (if present) would enable us to distinguish the pseudorandom function from a truly random function.

Formally, we prove this via a reduction. We use $\mathcal{A}$ to construct a distinguisher D for the pseudorandom function F. The distinguisher D is given oracle access to some function, and its goal is to determine whether this function is "pseudorandom" (i.e., equal to F_k for a randomly-chosen $k \leftarrow \{0,1\}^n$) or "random" (i.e., equal to f for a randomly-chosen $f \leftarrow \mathsf{Func}_n$). To do this, D emulates the CPA indistinguishability experiment for $\mathcal{A}$ in the manner described below, and observes whether $\mathcal{A}$ succeeds or not. If $\mathcal{A}$ succeeds then D guesses that its oracle must be a pseudorandom function, while if $\mathcal{A}$ does not succeed then D guesses that its oracle must be a random function. In detail:

Distinguisher D:
D is given input 1^n and access to an oracle $\mathcal{O} : \{0,1\}^n \to \{0,1\}^n$.

1. Run $\mathcal{A}(1^n)$. Whenever $\mathcal{A}$ queries its encryption oracle on a message m, answer this query in the following way:
   (a) Choose $r \leftarrow \{0,1\}^n$ uniformly at random.
   (b) Query $\mathcal{O}(r)$ and obtain response s'.
   (c) Return the ciphertext $\langle r, s' \oplus m \rangle$ to $\mathcal{A}$.
2. When $\mathcal{A}$ outputs messages $m_0, m_1 \in \{0,1\}^n$, choose a random bit $b \leftarrow \{0,1\}$ and then:
   (a) Choose $r \leftarrow \{0,1\}^n$ uniformly at random.
   (b) Query $\mathcal{O}(r)$ and obtain response s'.
   (c) Return the challenge ciphertext $\langle r, s' \oplus m_b \rangle$ to $\mathcal{A}$.
3. Continue answering any encryption oracle queries of $\mathcal{A}$ as before. Eventually, $\mathcal{A}$ outputs a bit b'. Output 1 if $b' = b$, and output 0 otherwise.

The key points are as follows:

1. If D's oracle is a pseudorandom function, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$. This holds because a key k is chosen at random and then every encryption is carried out by choosing a random r, computing $s' := F_k(r)$, and setting the ciphertext equal to $\langle r, s' \oplus m\rangle$, exactly as in Construction 3.24. Thus,

$$\Pr\left[D^{F_k(\cdot)}(1^n) = 1\right] = \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right],$$

   where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random in the above.

2. If D's oracle is a random function, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)$. This can be seen exactly as above, with the only difference being that a random function f is used instead of F_k. Thus,

$$\Pr\left[D^{f(\cdot)}(1^n) = 1\right] = \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right],$$

   where $f \leftarrow \mathsf{Func}_n$ is chosen uniformly at random in the above.

Combining Equations (3.4) and (3.5) with the above, we have that

$$\Pr\left[D^{F_k(\cdot)}(1^n) = 1\right] - \Pr\left[D^{f(\cdot)}(1^n) = 1\right] \geq \varepsilon(n) - \frac{q(n)}{2^n}$$

By the assumption that F is a pseudorandom function, it follows that $\varepsilon(n) - q(n)/2^n$ must be negligible. Since q is polynomial, this in turn implies that ε is negligible, and so Π is CPA secure, completing the proof. ∎

As discussed in Section 3.5, any CPA-secure fixed-length encryption scheme automatically yields a CPA-secure encryption scheme for messages of arbitrary length. Applying the approach discussed there to the fixed-length scheme we have just constructed, we have that an arbitrary-length message $m = m_1, \ldots, m_\ell$, where each m_i is an n-bit block, can be securely encrypted by computing

$$\langle r_1,\, F_k(r_1) \oplus m_1,\, r_2,\, F_k(r_2) \oplus m_2,\, \ldots,\, r_\ell,\, F_k(r_\ell) \oplus m_\ell \rangle.$$

(The scheme can handle messages whose length is not an exact multiple of n by truncation; we omit the details.) We have:

COROLLARY 3.26 *If F is a pseudorandom function, then the scheme sketched above is a private-key encryption scheme for arbitrary-length messages that has indistinguishable encryptions under a chosen-plaintext attack.*

Efficiency of Construction 3.24. The CPA-secure encryption scheme in Construction 3.24, and its extension to arbitrary-length messages in the corollary above, has the drawback that the length of the ciphertext is (at least) *double* the length of the plaintext. This is because each block of size n is encrypted using an n-bit random string which must be included as part of the ciphertext. In Section 3.6.4 we will show how the ciphertext length can be significantly reduced.

3.6.3 Pseudorandom Permutations and Block Ciphers

Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. We call F a keyed permutation if for every k, the function $F_k(\cdot)$ is *one-to-one* (since F is length-preserving, this implies that F_k is actually a bijection). We say that a keyed permutation is efficient if there is a polynomial-time algorithm computing $F_k(x)$ given k and x, *as well as* a polynomial-time algorithm computing $F_k^{-1}(x)$ given k and x.

The convention taken here that the lengths of the key, input, and output are all the same does not necessarily hold for constructions in practice. Rather, the input and output lengths — typically called the *block size* — are the same (which must be the case since it is a permutation), but the key length can be smaller or larger than the block size, depending on the construction. We assume they are all equal only to simplify notation.

We define what it means for an efficient keyed permutation F to be a *pseudorandom* permutation in a manner exactly analogous to Definition 3.23. The only change is that we now require that F_k (for a randomly-chosen k) be indistinguishable from a randomly-chosen *permutation* rather than a randomly-chosen function. Actually, this is merely an aesthetic decision since random permutations and (length-preserving) random functions are anyway indistinguishable using polynomially-many queries. Intuitively this is due to the fact that a random function f looks identical to a random permutation unless a distinct pair of values x and y are found for which $f(x) = f(y)$ (since in such a case the function cannot be a permutation). However, the probability of finding such points x, y using a polynomial number of queries is negligible. We leave a proof of the following for an exercise:

PROPOSITION 3.27 *If F is a pseudorandom permutation then it is also a pseudorandom function.*

If F is an efficient pseudorandom permutation then cryptographic schemes based on F might require honest parties to compute the inverse F_k^{-1} in addition to the permutation F_k itself. This potentially introduces new security concerns that are *not* covered by the fact that F is pseudorandom. In such a case, we may need to impose the stronger requirement that F_k be indistinguishable from a random permutation *even if the distinguisher is given oracle*

access to the inverse of the permutation. If F has this property, we call it a *strong* pseudorandom permutation. Formally:

DEFINITION 3.28 *Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, keyed permutation. We say that F is a* strong pseudorandom permutation *if for all probabilistic polynomial-time distinguishers D, there exists a negligible function* negl *such that:*

$$\left| \Pr[D^{F_k(\cdot), F_k^{-1}(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot), f^{-1}(\cdot)}(1^n) = 1] \right| \leq \mathsf{negl}(n),$$

where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random and f is chosen uniformly at random from the set of permutations on n-bit strings.

Of course, any strong pseudorandom permutation is also a pseudorandom permutation.

We noted earlier that a stream cipher can be modeled as a pseudorandom generator. The analogue for the case of strong pseudorandom permutations in practice is a *block cipher*. Unfortunately, it is often not stated in the literature that a block cipher is actually assumed to be a strong pseudorandom permutation. Explicitly modeling block ciphers in this way enables a formal analysis of many practical constructions that rely on block ciphers. These constructions include encryption schemes (as studied here), message authentication codes (to be studied in Chapter 4), authentication protocols, and more. Moreover, when proving security of a construction, it is important to specify whether the block cipher is being modeled as a pseudorandom permutation or a *strong* pseudorandom permutation. Although most block ciphers in use today are designed to satisfy the second, stronger requirement, a scheme that can be proven secure based on the former, weaker assumption may be preferable (since the requirements on the block cipher are potentially easier to satisfy).

As with stream ciphers, block ciphers themselves are *not* secure encryption schemes. Rather, they are building blocks that can be used to construct secure encryption schemes. For example, using a block cipher in Construction 3.24 yields a CPA-secure private-key encryption scheme. In contrast, an encryption scheme that works by just computing $c := F_k(m)$, where F_k is a strong pseudorandom permutation, is *not* CPA secure. This distinction between block ciphers as building blocks and encryption schemes that use block ciphers is of great importance and one that is too often missed.

While strong pseudorandom permutations are useful in the design and analysis of efficient cryptographic schemes, we will only use pseudorandom permutations (that are not necessarily strong) in the rest of this book.

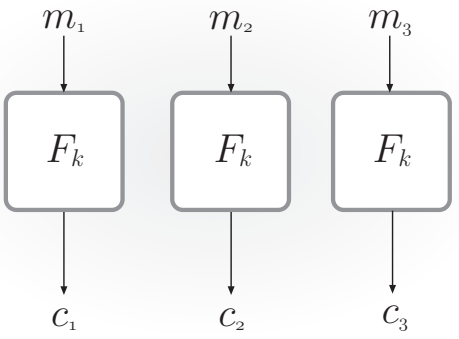

FIGURE 3.5: Electronic Code Book (ECB) mode.

3.6.4 Modes of Operation

A mode of operation is essentially a way of encrypting arbitrary-length messages using a block cipher (i.e., pseudorandom permutation). In Corollary 3.26 we have already seen one example of a mode of encryption, albeit one that is not very efficient in terms of the length of the ciphertext. In this section, we will see a number of modes of encryption with lower *ciphertext expansion* (defined to be the difference between the length of the ciphertext and the length of the message).

Before continuing, note that arbitrary-length messages can be unambiguously padded to a total length that is a multiple of any desired block size by appending a 1 followed by sufficiently-many 0s. We will therefore just assume that the length of the plaintext message is an exact multiple of the block size. Throughout this section, we will refer to a pseudorandom permutation/block cipher F with block length n, and will consider the encryption of messages consisting of ℓ blocks each of length n. We present four modes of operation and discuss their security.

Mode 1 — Electronic Code Book (ECB) mode. This is the most naive mode of operation possible. Given a plaintext message $m = m_1, m_2, \ldots, m_\ell$, the ciphertext is obtained by "encrypting" each block separately, where "encryption" here means a direct application of the pseudorandom permutation to the plaintext block. That is, $c = \langle F_k(m_1), F_k(m_2), \ldots, F_k(m_\ell)\rangle$; see Figure 3.5 for a graphic depiction with $\ell = 3$. Decryption is carried in the obvious way, using the fact that F_k^{-1} is efficiently computable.

The encryption process here is *deterministic* and therefore this mode of operation cannot possibly be CPA-secure (see the discussion following Definition 3.21). Even worse, *ECB-mode encryption does not have indistinguishable encryptions in the presence of an eavesdropper.* This is due to the fact that if the same block is repeated twice in the plaintext, this can be detected as a repeating block in the ciphertext. Thus, it is easy to distinguish an encryption of a plaintext that consists of two identical blocks from an encryption of

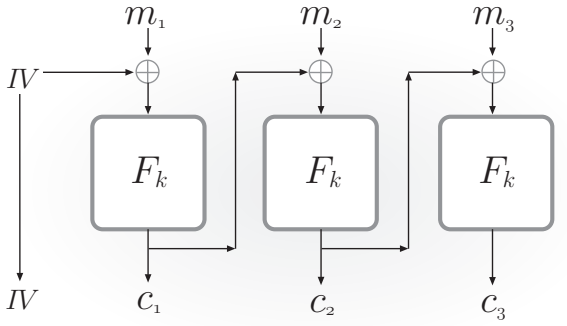

FIGURE 3.6: Cipher Block Chaining (CBC) mode.

a plaintext that consists of two different blocks. We stress that this is not just a "theoretical problem" and much information can be learned from viewing ciphertexts that are generated in this way. ECB mode should therefore never be used. (We include it for its historical significance only.)

Mode 2 — Cipher Block Chaining (CBC) mode. In this mode, a random initial vector (IV) of length n is first chosen. Then, each of the remaining ciphertext blocks is generated by applying the pseudorandom permutation to the XOR of the current plaintext block and the previous ciphertext block. That is, set $c_0 := IV$ and then, for $i = 1$ to ℓ, set $c_i := F_k(c_{i-1} \oplus m_i)$. The final ciphertext is $\langle c_0, c_1, \ldots, c_\ell \rangle$. (See Figure 3.6 for a graphical depiction.) We stress that the IV is sent in the clear as part of the ciphertext; this is crucial so that decryption can be carried out.

Importantly, encryption in CBC mode is probabilistic and it has been proven that if F is a pseudorandom permutation then CBC-mode encryption is CPA-secure. The main drawback of this mode is that encryption must be carried out sequentially because the ciphertext block c_{i-1} is needed in order to encrypt the plaintext block m_i. Thus, if parallel processing is available, CBC-mode encryption may not be the most efficient choice.

One may be tempted to think that it suffices to use a *distinct* IV (rather than a random IV) for every encryption; e.g., first use $IV = 1$ and then increment the IV by one each time. In Exercise 3.16, we ask you to show that this variant of CBC-mode encryption is not secure.

Mode 3 — Output Feedback (OFB) mode. The third mode we present here is called OFB. Essentially, this mode is a way of using a block cipher to generate a pseudorandom stream that is then XORed with the message. First, a random $IV \leftarrow \{0,1\}^n$ is chosen and a stream is generated from IV (independently of the plaintext message) in the following way: Define $r_0 := IV$, and set the ith block r_i of the stream to $r_i := F_k(r_{i-1})$. Then, each block of the plaintext is encrypted by XORing it with the appropriate block of the stream; that is, $c_i := m_i \oplus r_i$. (See Figure 3.7 for a graphical depiction.)

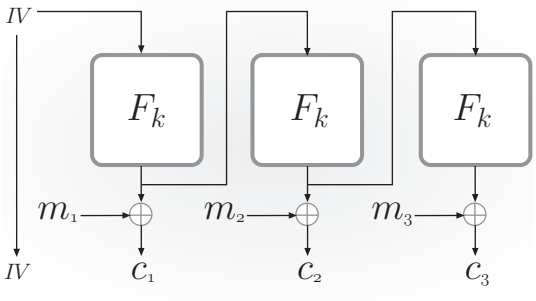

FIGURE 3.7: Output Feedback (OFB) mode.

As in CBC mode, the IV is included in the clear as part of the ciphertext in order to enable decryption; in contrast to CBC mode, here it is not required that F be invertible (in fact, it need not even be a permutation).

This mode is also probabilistic, and it can be shown that it too is a CPA-secure encryption scheme if F is a pseudorandom function. Here, both encryption and decryption must be carried out sequentially. On the other hand, this mode has the advantage that the bulk of the computation (namely, computation of the pseudorandom stream) can be done independently of the actual message to be encrypted. So, it may be possible to prepare a stream ahead of time using pre-processing, after which point the encryption of the plaintext (once it is known) is incredibly fast.

Mode 4 — Counter (CTR) mode. There are different variants of CTR-mode encryption; we describe the *randomized counter mode* here. As with OFB, counter mode can be viewed as a way of generating a pseudorandom stream from a block cipher. First, a random $IV \leftarrow \{0,1\}^n$ is chosen; here, this IV is often denoted ctr. Then, a stream is generated by computing $r_i := F_k(\mathsf{ctr} + i)$ (where ctr and i are viewed as integers and addition is performed modulo 2^n). Finally, the ith ciphertext block is computed as $c_i := r_i \oplus m_i$, and the IV is again sent as part of the ciphertext. See Figure 3.8 for a graphical depiction of this mode. Note once again that decryption does not require F to be invertible, or even a permutation.

Counter mode has a number of important properties. First and foremost, randomized counter mode (i.e., when ctr is chosen uniformly at random each time a message is encrypted) is CPA-secure, as will be proven below. Second, both encryption and decryption can be fully parallelized and, as with OFB mode, it is possible to generate the pseudorandom stream ahead of time, independently of the message. Finally, it is possible to decrypt the ith block of the ciphertext without decrypting anything else; this property is called *random access*. The above make counter mode a very attractive choice.

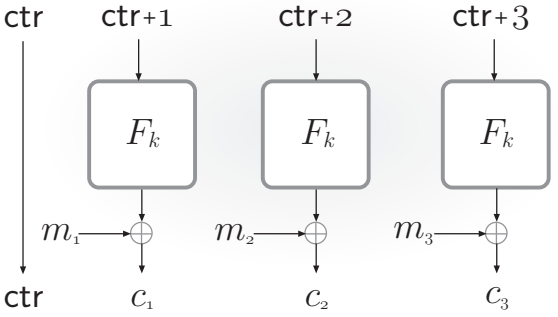

FIGURE 3.8: Counter (CTR) mode.

THEOREM 3.29 *If F is a pseudorandom function, then randomized counter mode (as described above) has indistinguishable encryptions under a chosen-plaintext attack.*

PROOF As in the proof of Theorem 3.25, we prove the present theorem by first showing that randomized counter mode is CPA-secure when a truly random function is used. We then prove that replacing the random function by a pseudorandom function cannot make the scheme insecure.

Let ctr^* denote the initial value ctr used when the challenge ciphertext is encrypted in the $\mathsf{PrivK}^{\mathsf{cpa}}$ experiment. Intuitively, when a random function f is used in randomized counter mode, security is achieved as long as each block c_i of the challenge ciphertext is encrypted using a value $\mathsf{ctr}^* + i$ that was never used by the encryption oracle in answering any of its queries. This is so because if $\mathsf{ctr}^* + i$ was never used to answer a previous encryption query, then the value $f(\mathsf{ctr}^* + i)$ is a completely random value, and so XORing this value with a block of the plaintext has the same effect as encrypting with the one-time pad. Proving that randomized counter mode is CPA-secure when using a random function thus boils down to bounding the probability that $\mathsf{ctr}^* + i$ was previously used.

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ denote the randomized counter mode encryption scheme, and let $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ be an encryption scheme that is identical to Π except that instead of using a pseudorandom permutation F with a randomly-chosen key, a truly random function is used instead. That is, $\widetilde{\mathsf{Gen}}(1^n)$ chooses a random function $f \leftarrow \mathsf{Func}_n$, and $\widetilde{\mathsf{Enc}}$ encrypts just like Enc except that f is used instead of F_k. (Of course, neither $\widetilde{\mathsf{Gen}}$ nor $\widetilde{\mathsf{Enc}}$ are efficient algorithms, but this does not matter for the purposes of defining an experiment involving $\widetilde{\Pi}$.) We now show that for every probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function negl such that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)\,. \tag{3.6}$$

Actually, we do not need to make any assumptions regarding the running time

(or computational power) of $\mathcal{A}$; it is sufficient to require only that $\mathcal{A}$ make a polynomial number of queries to its encryption oracle (with each query being a message of polynomial length), and in addition that $\mathcal{A}$ outputs m_0, m_1 of polynomial length.

Let q be a polynomial upper-bound on the number of oracle queries made by $\mathcal{A}$ as well as the maximum length of any such query and the maximum length of m_0, m_1. Fix some value n for the security parameter. Let ctr^* denote the initial value ctr used when the challenge ciphertext is encrypted, and let ctr_i denote the initial value ctr used when the encryption oracle answers the ith oracle query of $\mathcal{A}$. When the challenge ciphertext is encrypted, the function f is applied to the values $\mathsf{ctr}^* + 1, \ldots, \mathsf{ctr}^* + \ell^*$, where $\ell^* \leq q(n)$ is the length of m_0 and m_1. When the ith oracle query is answered, the function f is applied to the values $\mathsf{ctr}_i + 1, \ldots, \mathsf{ctr}_i + \ell_i$, where $\ell_i \leq q(n)$ is the length (in blocks) of the message whose encryption was requested. There are two cases to consider:

Case 1. *There do not exist any $i, j, j' \geq 1$ (with $j \leq \ell_i$ and $j' \leq \ell^*$) for which $\mathsf{ctr}_i + j = \mathsf{ctr}^* + j'$:* In this case, the values $f(\mathsf{ctr}^* + 1), \ldots, f(\mathsf{ctr}^* + \ell^*)$ used when encrypting the challenge ciphertext are independently and uniformly distributed since f was not applied to any of these inputs when encrypting any of the adversary's oracle queries. This means that the challenge ciphertext is computed by XORing a random stream of bits to the message m_b, and so the probability that $\mathcal{A}$ outputs $b' = b$ in this case is exactly $1/2$ (as in the case of the one-time pad).

Case 2. *There exist $i, j, j' \geq 1$ (with $j \leq \ell_i$ and $j' \leq \ell^*$) for which $\mathsf{ctr}_i + j = \mathsf{ctr}^* + j'$:* That is, the value used to encrypt block j of the ith encryption oracle query is the same as the value used to encrypt block j' of the challenge ciphertext. In this case $\mathcal{A}$ may easily determine which of its messages was encrypted to give the challenge ciphertext (since the adversary can deduce the value of $f(\mathsf{ctr}_i + j) = f(\mathsf{ctr}^* + j')$ from the answer to its ith oracle query).

Let us now analyze the probability that this occurs. The probability is maximized if ℓ^* and each ℓ_i are as large as possible, and so we assume that $\ell^* = \ell_i = q(n)$ for all i. Let $\mathsf{Overlap}_i$ denote the event that the sequence $\mathsf{ctr}_i + 1, \ldots, \mathsf{ctr}_i + q(n)$ overlaps the sequence $\mathsf{ctr}^* + 1, \ldots, \mathsf{ctr}^* + q(n)$, and let $\mathsf{Overlap}$ denote the event that $\mathsf{Overlap}_i$ occurs for some i. Since there are at most $q(n)$ oracle queries, a union bound gives

$$\Pr[\mathsf{Overlap}] \leq \sum_{i=1}^{q(n)} \Pr[\mathsf{Overlap}_i]. \tag{3.7}$$

Fixing ctr^*, event $\mathsf{Overlap}_i$ occurs exactly when ctr_i satisfies

$$\mathsf{ctr}^* + 1 - q(n) \leq \mathsf{ctr}_i \leq \mathsf{ctr}^* + q(n) - 1.$$

Since there are $2q(n) - 1$ values of ctr_i for which $\mathsf{Overlap}_i$ can occur, and ctr_i

is chosen uniformly at random from $\{0,1\}^n$, we see that

$$\Pr[\mathsf{Overlap}_i] = \frac{2q(n)-1}{2^n}.$$

Combined with Equation (3.7), this gives $\Pr[\mathsf{Overlap}] \leq 2q(n)^2/2^n$.

Given the above, we can easily bound the success probability of $\mathcal{A}$:

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)=1] &= \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)=1 \wedge \mathsf{Overlap}] \\ &\quad + \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)=1 \wedge \overline{\mathsf{Overlap}}] \\ &\leq \Pr[\mathsf{Overlap}] + \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)=1 \mid \overline{\mathsf{Overlap}}] \\ &\leq \frac{2q(n)^2}{2^n} + \frac{1}{2}.\end{aligned}$$

Since q is polynomial, $2q(n)^2/2^n$ is negligible, proving Equation (3.6). That is, the (imaginary) scheme $\widetilde{\Pi}$ is CPA-secure.

The next step in the proof is to show that this implies that Π (i.e., the scheme we are interested in) is CPA-secure; that is, that for any probabilistic polynomial-time $\mathcal{A}$ there exists a negligible function negl' such that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)=1\right] \leq \frac{1}{2} + \mathsf{negl}'(n)\,.$$

Intuitively, this is because replacing the random function f used in $\widetilde{\Pi}$ by the pseudorandom function F_k used in Π should have "no effect" as far as a polynomial-time adversary is concerned. Of course, this intuition should be rigorously proved. This proof is very similar to the analogous step in the proof of Theorem 3.25, and so is left as an exercise. ■

Block length and security. All the above modes (with the exception of ECB that is anyway not secure) use a random IV. The IV has the effect of randomizing the encryption process, and ensures that (with high probability) the block cipher is always evaluated on a *new* input that was never used before. This is important because, as we have seen in the proofs of Theorem 3.25 and Theorem 3.29, if an input to the block cipher is used more than once then security can be violated. (E.g., in the case of counter mode, the same pseudorandom string will be XORed with two different plaintext blocks.) Interestingly, this shows that it is not only the *key length* of a block cipher that is important in evaluating its security, but also its *block length*. For example, say we use a block cipher with a 64-bit block length. We showed in the proof of Theorem 3.29 that, in randomized counter mode, even if a completely random function with this block length is used (i.e., even if the block cipher is "perfect"), an adversary can achieve success probability roughly $\frac{1}{2} + \frac{q^2}{2^{63}}$

in a chosen-plaintext attack when it makes q queries to its encryption oracle, each q blocks long. Although this is asymptotically negligible (when the block length grows as a function of the security parameter n), security no longer holds in any practical sense (for this particular block length) when $q \approx 2^{30}$. Depending on the application, one may want to switch to a block cipher having a larger block length (2^{30} is only one gigabyte, which is not much considering today's storage needs).

Other modes of operation. In recent years, many different modes of operation have been introduced, each offering its own unique advantages in some setting. Nevertheless, in general, CBC, OFB, and CTR modes suffice for most applications where CPA-security is needed.

Modes of encryption and message tampering. In many texts on cryptography, modes of operation are also compared based on how well they protect against adversarial modifications of the ciphertext. We do *not* include such a comparison here because the issue of *message integrity* or *message authentication* should be dealt with separately from encryption, and we do so in the next chapter. None of the above modes achieve message integrity in the sense we will define there. Further discussion is given in the next chapter.

Stream ciphers versus block ciphers. As we have seen here for the OFB and counter modes, it is possible to work in "stream-cipher mode" using a block-cipher (i.e., generating a stream of pseudorandom bits and XORing this stream with the plaintext). Furthermore, a block cipher can be used to generate multiple (independent) pseudorandom streams, while (in general) a stream cipher is limited to generating a single such stream. This begs the question: which is preferable, a block cipher or a stream cipher? The only advantage of stream ciphers is their relative efficiency, though this gain may only be a factor of two unless one is using severely resource-constrained devices such as PDAs or cell phones.[11] On the other hand, stream ciphers appear to be much less well understood (in practice) than block ciphers. There are a number of excellent block ciphers that are efficient and believed to be highly secure (we will study two of these in Chapter 5). In contrast, stream ciphers seem to be broken more often, and confidence in their security is lower. It is also more likely that stream ciphers will be misused in such a way that the same pseudorandom stream will be used twice. We therefore recommend using block ciphers unless for some reason this is not possible.

[11] In particular, estimates from [42] indicate that on a typical home PC the stream cipher RC4 is only about twice as fast as the block cipher AES, measured in terms of bits/second.

3.7 Security Against Chosen-Ciphertext Attacks (CCA)

Until now, we have defined security against two types of adversaries: a passive adversary that only eavesdrops, and an active adversary that carries out a chosen-plaintext attack. A third type of attack, called a *chosen-ciphertext attack*, is even more powerful than these two. In a chosen-ciphertext attack, we provide the adversary not only with the ability to encrypt messages of its choice (as in a chosen-plaintext attack), but also with the ability to *decrypt* ciphertexts of its choice (with one exception discussed later). Formally, we give the adversary access to a *decryption oracle* in addition to an encryption oracle. We present the formal definition and defer further discussion until afterward.

Consider the following experiment for any private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, adversary $\mathcal{A}$, and value n for the security parameter.

The CCA indistinguishability experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$:

1. *A key k is generated by running $\mathsf{Gen}(1^n)$.*
2. *The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\mathsf{Enc}_k(\cdot)$ and $\mathsf{Dec}_k(\cdot)$. It outputs a pair of messages m_0, m_1 of the same length.*
3. *A random bit $b \leftarrow \{0,1\}$ is chosen, and then a ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$. We call c the* challenge ciphertext.
4. *The adversary $\mathcal{A}$ continues to have oracle access to $\mathsf{Enc}_k(\cdot)$ and $\mathsf{Dec}_k(\cdot)$, but is not allowed to query the latter on the challenge ciphertext itself. Eventually, $\mathcal{A}$ outputs a bit b'.*
5. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.*

DEFINITION 3.30 *A private-key encryption scheme Π has* indistinguishable encryptions under a chosen-ciphertext attack *(or is* CCA-secure*) if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over all random coins used in the experiment.

In the experiment above, the adversary's access to the decryption oracle is unlimited *except* for the restriction that the adversary may not request decryption of the challenge ciphertext itself. This restriction is necessary or else there is clearly no hope for any encryption scheme to satisfy Definition 3.30.

At this point you may be wondering if chosen-ciphertext attacks realistically model any real-world attack. As in the case of a chosen-plaintext attack, we do not expect honest parties to decrypt arbitrary ciphertexts of an adversary's choice. Nevertheless, there may be scenarios where an adversary might be able to *influence* what gets decrypted, and learn some partial information about the result:

1. In the case of Midway (see Section 3.5) it is conceivable that the US cryptanalysts might also have tried to send encrypted messages to the Japanese and then monitor their behavior. Such behavior (e.g., movement of forces and the like) could have provided important information about the underlying plaintext.

2. Imagine a user communicating with their bank, where all communication is encrypted. If this communication is not authenticated, then an adversary may be able to send certain ciphertexts on behalf of the user; the bank will decrypt these ciphertexts, and the adversary may learn something about the result. For example, if a ciphertext corresponds to an *ill-formed* plaintext (e.g., a gibberish message, or simply one that is not formatted correctly), the adversary may be able to deduce this from the bank's reaction (i.e., the pattern of subsequent communication).

3. Encryption is often used in higher-level protocols; e.g., an encryption scheme might be used as part of an authentication protocol where one party sends a ciphertext to the other, who decrypts it and returns the result. In this case, one of the honest parties may act exactly like a decryption oracle and so the scheme must be CCA secure.

Insecurity of the schemes we have studied. None of the encryption schemes we have seen is CCA-secure. We will demonstrate this for Construction 3.24, where encryption is carried out as $\mathsf{Enc}_k(m) = \langle r, F_k(r) \oplus m \rangle$. The fact that this scheme is not CCA-secure can be easily demonstrated as follows. An adversary $\mathcal{A}$ running in the CCA indistinguishability experiment can choose $m_0 = 0^n$ and $m_1 = 1^n$. Then, upon receiving a ciphertext $c = \langle r, s \rangle$, the adversary $\mathcal{A}$ can flip the first bit of s and ask for a decryption of the resulting ciphertext c'. Since $c' \neq c$, this query is allowed, and the decryption oracle answers with either 10^{n-1} (in which case it is clear that $b = 0$) or 01^{n-1} (in which case $b = 1$). This example demonstrates why CCA-security is so stringent. Specifically, any encryption scheme that allows ciphertexts to be manipulated in any "logical way" cannot be CCA-secure. Thus, CCA-security actually implies a very important property called *non-malleability*. Loosely speaking, a non-malleable encryption scheme has the property that if the adversary tries to modify a given ciphertext, the result is either an illegal ciphertext or one that encrypts a plaintext having no relation to the original one. We leave for an exercise the demonstration that none of the modes of encryption we have seen is CCA-secure.

Constructing a CCA-secure encryption scheme. We show how to construct a CCA-secure encryption scheme in Section 4.8. The construction is presented there because it uses tools that we develop in Chapter 4.

References and Additional Reading

The modern computational approach to cryptography was initiated in a ground-breaking paper by Goldwasser and Micali [69]. That paper introduced the notion of semantic security, and showed how this goal could be achieved in the setting of public-key encryption (see Chapters 9 and 10). The paper also proposed the notion of indistinguishability (i.e., Definition 3.8), and showed that it implies semantic security. The converse was shown in [104].

Formal definitions of security against chosen-plaintext attacks were given by Luby [96] and Bellare et al. [9]. Chosen-ciphertext attacks (in the context of public-key encryption) were first formally defined by Naor and Yung [107] and Rackoff and Simon [121], and were considered also in [50] and [9]. See [86] for other notions of security for private-key encryption.

Blum and Micali [23] introduced the notion of pseudorandom generators, and proved their existence based on a specific number-theoretic assumption. In the same work, Blum and Micali also pointed out the connection between pseudorandom generators and private-key encryption as in Construction 3.15. The definition of pseudorandom generators given by Blum and Micali is different from the definition we use in this book (Definition 3.14); the latter definition originates in the work of Yao [149] who showed equivalence of the two formulations. Yao also showed constructions of pseudorandom generators based on general assumptions, and we will explore this result in Chapter 6.

Pseudorandom functions were defined and constructed by Goldreich et al. [67], and their application to encryption was demonstrated in subsequent work by the same authors [66]. Pseudorandom permutations and strong pseudorandom permutations were studied by Luby and Rackoff [97]. Of course, block ciphers had been used for many years before they began to be studied in the theoretical sense initiated by these works.

The ECB, CBC, and OFB modes of operation (as well as CFB mode, an additional mode of operation not covered here) were standardized along with the DES block cipher [108]. The CBC and CTR modes of encryption were proven CPA-secure in [9]. For more recent modes of encryption, see `http://csrc.nist.gov/CryptoToolkit`. A good but somewhat outdated overview of stream ciphers used in practice can be found in [99, Chapter 6]. The RC4 stream cipher is discussed in [125] and an accessible discussion of recent attacks and their ramifications can be found in [56].

Exercises

3.1 Prove Proposition 3.6.

3.2 The best algorithm known today for finding the prime factors of an n-bit number runs in time $2^{c \cdot n^{\frac{1}{3}} (\log n)^{\frac{2}{3}}}$. Assuming 4Ghz computers and $c = 1$ (and that the units of the given expression are clock cycles), estimate the size of numbers that cannot be factored for the next 100 years.

3.3 Prove that Definition 3.8 cannot be satisfied if Π can encrypt arbitrary-length messages and the adversary is *not* restricted to output equal-length messages in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$.

> **Hint:** Let $q(n)$ be a polynomial upper-bound on the length of the ciphertext when Π is used to encrypt a single bit. Then consider an adversary who outputs $m_0 \in \{0,1\}$ and a random $m_1 \in \{0,1\}^{q(n)+2}$.

3.4 Say $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is such that for $k \in \{0,1\}^n$, algorithm Enc_k is only defined for messages of length at most $\ell(n)$ (for some polynomial ℓ). Construct a scheme satisfying Definition 3.8 even when the adversary is *not* restricted to output equal-length messages in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$.

3.5 Prove the equivalence of Definition 3.8 and Definition 3.9.

3.6 Let G be a pseudorandom generator where $|G(s)| > 2 \cdot |s|$.

(a) Define $G'(s) \stackrel{\text{def}}{=} G(s0^{|s|})$. Is G' necessarily a pseudorandom generator?

(b) Define $G'(s) \stackrel{\text{def}}{=} G(s_1 \cdots s_{n/2})$, where $s = s_1 \cdots s_n$. Is G' necessarily a pseudorandom generator?

3.7 Assuming the existence of a pseudorandom function, prove that there exists an encryption scheme that has indistinguishable multiple encryptions in the presence of an eavesdropper (i.e., is secure with respect to Definition 3.18), but is not CPA-secure (i.e., is not secure with respect to Definition 3.21).

> **Hint:** The scheme need not be "natural". You will need to use the fact that in a chosen-plaintext attack the adversary can choose its queries to the encryption oracle *adaptively*.

3.8 Prove *unconditionally* the existence of an efficient pseudorandom function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ where the input length is *logarithmic* in the security parameter (and the key has length polynomial in the security parameter).

> **Hint:** Implement a random function with logarithmic input length.

3.9 Present a construction of a variable output-length pseudorandom generator from any pseudorandom function. Prove that your construction satisfies Definition 3.17.

3.10 Let G be a pseudorandom generator and define $G'(s)$ to be the output of G truncated to n bits (where $|s| = n$). Prove that the function $F_k(x) = G'(k) \oplus x$ is not pseudorandom.

3.11 Prove Proposition 3.27 (i.e., prove that any pseudorandom permutation is also a pseudorandom function).

Hint: Show that in polynomial time, a random permutation cannot be distinguished from a random function (use the results of Appendix A.4).

3.12 Define a notion of perfect secrecy against a chosen-plaintext attack via the natural adaptation of Definition 3.21. Show that the definition cannot be achieved.

3.13 Assume that F is a pseudorandom permutation. Show that there exists a function F' that is a pseudorandom permutation but is *not* a strong pseudorandom permutation.

Hint: Construct F' such that $F'_k(k) = 0^{|k|}$.

3.14 Let F be a pseudorandom permutation, and define a fixed-length encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ as follows: On input $m \in \{0,1\}^{n/2}$ and key $k \in \{0,1\}^n$, algorithm Enc chooses a random string $r \leftarrow \{0,1\}^{n/2}$ of length $n/2$ and computes $c := F_k(r \| m)$.

Show how to decrypt, and prove that this scheme is CPA-secure for messages of length $n/2$. (If you are looking for a real challenge, prove that this scheme is CCA-secure if F is a *strong* pseudorandom permutation.) What are the advantages and disadvantages of this construction as compared to Construction 3.24?

3.15 Let F be a pseudorandom function, and G a pseudorandom generator with expansion factor $\ell(n) = n+1$. For each of the following encryption schemes, state whether the scheme has indistinguishable encryptions in the presence of an eavesdropper and whether it is CPA-secure. In each case, the shared key is a random $k \in \{0,1\}^n$.

(a) To encrypt $m \in \{0,1\}^{2n+2}$, parse m as $m_1 \| m_2$ with $|m_1| = |m_2|$ and send $\langle G(k) \oplus m_1,\ G(k+1) \oplus m_2 \rangle$.

(b) To encrypt $m \in \{0,1\}^{n+1}$, choose a random $r \leftarrow \{0,1\}^n$ and send $\langle r, G(r) \oplus m \rangle$.

(c) To encrypt $m \in \{0,1\}^n$, send $m \oplus F_k(0^n)$.

(d) To encrypt $m \in \{0,1\}^{2n}$, parse m as $m_1 \| m_2$ with $|m_1| = |m_2|$, then choose $r \leftarrow \{0,1\}^n$ at random, and send $\langle r,\ m_1 \oplus F_k(r),\ m_2 \oplus F_k(r+1) \rangle$.

3.16 Consider a variant of CBC-mode encryption where the sender simply increments the IV by 1 each time a message is encrypted (rather than choosing IV at random each time). Show that the resulting scheme is *not* CPA-secure.

3.17 Present formulas for decryption of all the different modes of encryption we have seen. For which modes can decryption be parallelized?

3.18 Complete the proof of Theorem 3.29.

3.19 Let F be a pseudorandom function such that for $k \in \{0,1\}^n$ the function F_k maps $\ell_{in}(n)$-bit inputs to $\ell_{out}(n)$-bit outputs. (Throughout this chapter, we have assumed $\ell_{in}(n) = \ell_{out}(n) = n$.)

(a) Consider implementing counter mode encryption using an F of this form. For which functions ℓ_{in}, ℓ_{out} is the resulting encryption scheme CPA-secure?

(b) Consider implementing counter mode encryption using an F as above, but only for *fixed-length* messages of length $\ell(n)$ (which is always an integer multiple of $\ell_{out}(n)$). For which $\ell_{in}, \ell_{out}, \ell$ is the scheme CPA-secure? For which $\ell_{in}, \ell_{out}, \ell$ does the scheme have indistinguishable encryptions in the presence of an eavesdropper?

3.20 For a function $g : \{0,1\}^n \to \{0,1\}^n$, let $g^{\$}(\cdot)$ be an oracle that, on input 1^n, chooses $r \leftarrow \{0,1\}^n$ uniformly at random and returns $(r, g(r))$. We say a keyed function F is a weak pseudorandom function if for all PPT algorithms D, there exists a negligible function negl such that:

$$\left| \Pr[D^{F_k^{\$}(\cdot)}(1^n) = 1] - \Pr[D^{f^{\$}(\cdot)}(1^n) = 1] \right| \leq \mathsf{negl}(n),$$

where $k \leftarrow \{0,1\}^n$ and $f \leftarrow \mathsf{Func}_n$ are chosen uniformly at random.

(a) Prove that if F is pseudorandom then it is weakly pseudorandom.

(b) Let F' be a pseudorandom function, and define

$$F_k(x) = \begin{cases} F'_k(x) & \text{if } x \text{ is even} \\ F'_k(x+1) & \text{if } x \text{ is odd} \end{cases}.$$

Prove that F is weakly pseudorandom, but *not* pseudorandom.

(c) Is counter-mode encryption instantiated using a weak pseudorandom function F necessarily CPA-secure? Does it necessarily have indistinguishable encryptions in the presence of an eavesdropper? Prove your answers.

(d) Construct a CPA-secure encryption scheme based on a weak pseudorandom function.

Hint: One of the constructions in this chapter will work.

3.21 Let $\Pi_1 = (\mathsf{Gen}_1, \mathsf{Enc}_1, \mathsf{Dec}_1)$ and $\Pi_2 = (\mathsf{Gen}_2, \mathsf{Enc}_2, \mathsf{Dec}_2)$ be two encryption schemes for which it is known that at least one is CPA-secure. The problem is that you don't know which one is CPA-secure and which one may not be. Show how to construct an encryption scheme Π that is guaranteed to be CPA-secure as long as at least one of Π_1 or Π_2 is CPA-secure. Try to provide a full proof of your answer.

> **Hint:** Generate two plaintext messages from the original plaintext so that knowledge of either one of the parts reveals nothing about the plaintext, but knowledge of both does yield the original plaintext.

3.22 Show that the CBC, OFB, and counter modes of encryption do not yield CCA-secure encryption schemes (regardless of F).

Chapter 4

Message Authentication Codes and Collision-Resistant Hash Functions

4.1 Secure Communication and Message Integrity

One of the most basic goals of cryptography is to enable parties to communicate over an *open communication channel* in a secure way. But what does "secure communication" entail? In Chapter 3 we showed that it is possible to obtain *private* communication over an open channel. That is, we showed how encryption can be used to prevent an eavesdropper (or possibly a more active adversary) from learning anything about the content of messages sent over an unprotected communication channel. However, not all security concerns are related to the ability or inability of an adversary to learn information about messages being sent. In many cases, it is of equal or greater importance to guarantee *message integrity* (or *message authentication*) in the sense that each party should be able to identify when a message it receives was exactly the message sent by the other party. For example, consider the case that a large supermarket chain sends an email request to purchase 10,000 crates of soda from a supplier. Upon receiving such a request, the supplier has to consider the following:

1. Is the order authentic? That is, did the supermarket chain really issue an order, or was the order issued by an adversary who spoofed the email address of the supermarket (something that is remarkably easy to do)?

2. Even if it can be assured that an order was issued by the supermarket, the supplier must still ask whether the details of the order are exactly those intended by the supermarket, or whether the order was changed en route by an adversarial entity (e.g., a malicious router).

The order itself is not secret and therefore the question of privacy does not arise here at all. Rather, the problem is purely one of message integrity.

In general, one cannot rely on the integrity of communication without taking specific measures to ensure it. Indeed, any unprotected online purchase order, online banking operation, email, or SMS message cannot, in general, be trusted to have originated from the claimed source. Unfortunately, people

are in general trusting and thus information like the caller-ID or an email return address are taken to be "proofs of origin" in many cases (even though they are relatively easy to forge). This leaves the door open to potentially damaging adversarial attacks.

In this chapter we will show how to use cryptographic techniques to prevent the *undetected* tampering of messages that are sent over an open communication line, and thus achieve message integrity in the sense described above. Note that we cannot prevent adversarial tampering of messages altogether, as this can only be defended against at the physical level. Instead, what we will guarantee is that any such tampering will be detected by the honest parties.

4.2 Encryption vs. Message Authentication

Just as the goals of privacy and message integrity are different, so are the techniques and tools for achieving them. Unfortunately, privacy and integrity are often confused and unnecessarily intertwined, so let us be clear up-front: encryption does *not* (in general) provide any integrity, and encryption should never be used with the intent of achieving message authentication. (Naturally, encryption may be used *in conjunction with* other techniques for achieving message authentication, something we will return to at the end of this chapter.)

One might mistakenly think, at first, that encryption immediately solves the problem of message authentication. (In fact, this is a common error.) This is due to the fuzzy (and incorrect) reasoning that since a ciphertext completely hides the contents of the message, an adversary cannot possibly modify an encrypted message in any meaningful way. Despite its intuitive appeal, this reasoning is completely false. We illustrate this point by showing that all the encryption schemes we have seen thus far do not provide message integrity.

Encryption using stream ciphers. First, consider the case that a message m is encrypted using a stream cipher. That is, $\mathsf{Enc}_k(m)$ computes the ciphertext $c := G(k) \oplus m$, where G is a pseudorandom generator. Ciphertexts in this case are very easy to manipulate. Specifically, flipping any bit in the ciphertext c results in the same bit being flipped in the message that is recovered upon decryption. Thus, given a ciphertext c that encrypts a message m, it is possible to generate a modified ciphertext c' such that $m' = \mathsf{Dec}_k(c')$ is the same as m but with one (or more) of the bits flipped. This simple attack can have severe consequences. As a simple example, consider a user encrypting some dollar amount they want to transfer from their bank account, where this amount is represented in binary. Flipping the least significant bit has the effect of changing this amount by only \$1, but flipping the 11th least significant

bit changes the amount by more than $1,000! Interestingly, the adversary in this example does not learn whether it is increasing or decreasing the initial amount (i.e., whether it is flipping a 0 to a 1 or vice versa). Furthermore, the existence of this attack does not contradict the privacy of the encryption scheme (in the sense of Definition 3.8). In fact, the exact same attack applies to the one-time pad encryption scheme, showing that even perfect secrecy is not sufficient to ensure the most basic level of message integrity.

Encryption using block ciphers. The aforementioned attack utilizes the fact that flipping a single bit in a ciphertext keeps the underlying plaintext unchanged except for the corresponding bit (which is also flipped). The same attack applies to the OFB and counter mode encryption schemes, which also encrypt messages by XORing them with a pseudorandom stream (albeit one that changes each time a message is encrypted). We thus see that even using CPA-secure encryption is not enough to prevent message tampering.

One may hope that attacking ECB- or CBC-mode encryption would be more difficult since decryption in these cases involves inverting a strong pseudorandom function F, and we expect that $F_k^{-1}(x)$ and $F_k^{-1}(x')$ will be completely uncorrelated even if x and x' differ in only a single bit. (Of course ECB mode does not even guarantee the most basic notion of privacy, but that does not matter for the present discussion.) Nevertheless, single-bit modifications of a ciphertext still cause reasonably predictable changes in the plaintext. For example, when using ECB mode, flipping a bit in the ith block of the ciphertext affects only the ith block of the plaintext — all other blocks remain unchanged. Though the effect on the ith block of the plaintext may be impossible to predict, changing that one block (while leaving everything else unchanged) may represent a harmful attack. Similarly, when using CBC mode, flipping the jth bit of the IV changes only the jth bit of the first message block m_1 (since $m_1 := F_k(c_1) \oplus IV'$, where IV' is the modified IV); all plaintext blocks other than the first remain unchanged (since the ith block of the plaintext is computed as $m_i := F_k^{-1}(c_i) \oplus c_{i-1}$, and blocks c_i, c_{i-1} have not been modified). To make things worse, the order of blocks in ECB can be changed (without garbling any block), and the first block of a CBC-encrypted message can be tampered with arbitrarily as with a stream cipher. This integrity attack on CBC-mode is particularly troublesome because the first block of a message often contains highly important header information.

Finally, we point out that all encryption schemes we have seen thus far have the property that *every* possible ciphertext (perhaps satisfying some length constraint) corresponds to *some* valid message. So it is trivial for an adversary to "spoof" a message on behalf of one of the communicating parties — by sending some arbitrary ciphertext — even if the adversary has no idea what the underlying message will be. As we will see when we formally define authenticated communication, even an attack of this sort should be ruled out.

4.3 Message Authentication Codes – Definitions

As we have seen, encryption does not solve the problem of message authentication. Rather, an additional mechanism is needed that will enable the communicating parties to know whether or not a message was tampered with. Such mechanisms are called message authentication codes.

The aim of a message authentication code is to prevent an adversary from modifying a message sent by one party to another, without the parties detecting that a modification has been made. As in the case of encryption, this is only possible if the communicating parties have some secret that the adversary does not know (otherwise nothing can prevent an adversary from impersonating the party sending the message). Here, we will continue to consider the private-key setting where the parties share the same secret key.

The syntax of a message authentication code. Before formally defining security of a message authentication code (MAC), we first define what a MAC is and how it is used. Two users who wish to communicate in an authenticated manner begin by generating and sharing a secret key k in advance of their communication. When one party wants to send a message m to the other, she computes a MAC tag (or simply a tag) t based on the message and the shared key, and sends the message m along with the tag t to the other party. The tag is computed using a *tag-generation algorithm* that will be denoted by Mac; rephrasing what we have already said, the sender of a message m computes $t \leftarrow \mathsf{Mac}_k(m)$ and transmits (m, t) to the receiver. Upon receiving (m, t), the second party *verifies* whether t is a valid tag on the message m (with respect to the shared key) or not. This is done by running a *verification algorithm* Vrfy that takes as input the shared key as well as a message m and a tag t, and indicates whether the given tag is valid. Formally:

DEFINITION 4.1 *A* message authentication code (*or* MAC) *is a tuple of probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *such that:*

1. *The* key-generation algorithm Gen *takes as input the security parameter* 1^n *and outputs a key* k *with* $|k| \geq n$.

2. *The* tag-generation algorithm Mac *takes as input a key* k *and a message* $m \in \{0,1\}^*$*, and outputs a tag* t*. Since this algorithm may be randomized, we write this as* $t \leftarrow \mathsf{Mac}_k(m)$.

3. *The* verification algorithm Vrfy *takes as input a key* k*, a message* m*, and a tag* t*. It outputs a bit* b*, with* $b = 1$ *meaning* valid *and* $b = 0$ *meaning* invalid*. We assume without loss of generality that* Vrfy *is deterministic, and so write this as* $b := \mathsf{Vrfy}_k(m, t)$.

It is required that for every n, every key k output by $\mathsf{Gen}(1^n)$, and every $m \in \{0,1\}^*$, it holds that $\mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m)) = 1$.

If $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ is such that for every k output by $\mathsf{Gen}(1^n)$, algorithm Mac_k is only defined for messages $m \in \{0,1\}^{\ell(n)}$ (and Vrfy_k outputs 0 for any $m \notin \{0,1\}^{\ell(n)}$), then we say that $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ is a fixed-length MAC *for messages of length $\ell(n)$.*

As with private-key encryption, it is almost always the case that $\mathsf{Gen}(1^n)$ chooses $k \leftarrow \{0,1\}^n$ uniformly at random.

Security of message authentication codes. We now define the notion of security for message authentication codes. (Thankfully, in contrast to the case of private-key encryption, there is only one generally-accepted definition of security in this context.) The intuitive idea behind the definition of security is that no polynomial-time adversary should be able to generate a valid tag on any "new" message that was not previously sent (and authenticated) by one of the communicating parties.

As with any security definition, to formalize this notion we have to define both the adversary's power as well as what should be considered a "break". As usual, we consider only probabilistic polynomial-time adversaries[1] and so the real question with regard to the power of the adversary is how we model the adversary's interaction with the communicating parties. In the setting of message authentication, an adversary observing the communication between the honest parties will be able to see all the messages sent by these parties along with their corresponding MAC tags. The adversary may also be able to influence the *content* of these messages, either indirectly (if external actions of the adversary affect the messages sent by the parties), or directly. As an example of the latter, consider the case where the adversary is the personal assistant of one of the parties and has significant control over what messages this party sends.

To formally model the above possibilities, we allow the adversary to request MAC tags for *any* messages of its choice. Formally, we give the adversary access to a *MAC oracle* $\mathsf{Mac}_k(\cdot)$; the adversary can submit any message m that it likes to this oracle, and is given in return a tag $t \leftarrow \mathsf{Mac}_k(m)$.

We will consider it a "break" of the scheme if the adversary is able to output any message m along with a tag t such that: **(1)** t is a valid tag on the message m (i.e., $\mathsf{Vrfy}_k(m,t) = 1$) and **(2)** the adversary had not previously requested a MAC tag on the message m (i.e., from its oracle). Adversarial success in the first condition means that, in the real world, if the adversary were to send (m,t) to one of the honest parties, then this party would be

[1] As noted in the "References and Additional Reading" section of Chapter 2, notions of perfectly-secure message authentication — where no computational restrictions are placed on the adversary — can also be considered. As in the case of perfectly-secret encryption, however, perfectly-secure MACs suffer from severe bounds on their efficiency that limit their usefulness in practice.

mistakenly fooled into thinking that m originated from the legitimate party (since $\mathsf{Vrfy}_k(m,t)=1$). The second condition is required because it is always possible for the adversary to just copy a message and MAC tag that was previously sent by the legitimate parties (and, of course, these would be accepted). Such an adversarial attack is called a *replay attack* and is not considered a "break" of the message authentication code. This does *not* mean that replay attacks are not a security concern; they are, and we will have more to say about this further below.

A MAC satisfying the level of security specified above is said to be *existentially unforgeable under an adaptive chosen-message attack*. "Existential unforgeability" refers to the fact that the adversary must not be able to forge a valid tag on *any* message, and "adaptive chosen-message attack" refers to the fact that the adversary is able to obtain MAC tags on any message it likes, where these messages may be chosen adaptively during its attack.

Toward the formal definition, consider the following experiment for a message authentication code $\Pi=(\mathsf{Gen},\mathsf{Mac},\mathsf{Vrfy})$, an adversary $\mathcal{A}$, and value n for the security parameter:

The message authentication experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$**:**

1. *A random key k is generated by running* $\mathsf{Gen}(1^n)$.
2. *The adversary $\mathcal{A}$ is given input* 1^n *and oracle access to* $\mathsf{Mac}_k(\cdot)$. *The adversary eventually outputs a pair (m,t). Let $\mathcal{Q}$ denote the set of all queries that $\mathcal{A}$ asked to its oracle.*
3. *The output of the experiment is defined to be 1 if and only if* **(1)** $\mathsf{Vrfy}_k(m,t)=1$ *and* **(2)** $m\notin\mathcal{Q}$.

We define a MAC to be secure if no efficient adversary can succeed in the above experiment with non-negligible probability.

DEFINITION 4.2 *A message authentication code* $\Pi=(\mathsf{Gen},\mathsf{Mac},\mathsf{Vrfy})$ *is* existentially unforgeable under an adaptive chosen-message attack*, or just* secure*, if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1]\leq\mathsf{negl}(n).$$

Is the definition *too* strong? The above definition is rather strong, in two respects. First, the adversary is allowed to request a MAC tag for *any* message of its choice. Second, the adversary is considered to have "broken" the scheme if it can output a valid tag on *any* previously-unauthenticated message. One might object that both of these components of the definition are unrealistic and overly strong: in "real-world" usage of a MAC, the honest parties would only authenticate "meaningful" messages (over which the adversary might have only limited control), and similarly it should only be considered a breach of security if the adversary can forge a valid tag on a "meaningful" message. Why not tailor the definition to capture this?

The crucial point is that what constitutes a meaningful message is entirely *application-dependent*. While some applications of a MAC may only ever authenticate English-text messages, other applications may authenticate spreadsheet files, others database entries, and others raw data. Protocols may also be designed where *anything* will be authenticated — in fact, certain protocols for entity authentication do exactly this. By making the definition of security for MACs as strong as possible, we ensure that secure MACs are broadly applicable for a wide range of purposes, without having to worry about compatibility of the MAC with the semantics of the application.

Replay attacks. We emphasize that the above definition, and message authentication codes in general, offer no protection against *replay attacks* in which a previously-sent message (and its MAC tag) are replayed to one of the honest parties. Nevertheless, replay attacks are a serious concern. Consider the following scenario: a user Alice sends her bank an order to transfer \$1,000 from her account to Bob's account. In doing so, Alice computes a MAC tag and appends it to the message so that the bank knows that the message is authentic. If the MAC is secure, Bob will be unable to intercept the message and change the amount to \$10,000 (because this would involve forging a valid tag on a previously-unauthenticated message). However, nothing prevents Bob from intercepting Alice's message and replaying it *ten times* to the bank. If the bank accepts each of these messages, the net effect is that \$10,000 will be transferred to Bob's account rather than the desired \$1,000.

Despite the real threat due to replay attacks, a MAC inherently *cannot* protect against such attacks since the definition of a MAC (Definition 4.1) does not incorporate any notion of *state* into the verification algorithm (and so every time a valid pair (m, t) is presented to the verification algorithm, it will always output 1). Rather, protection against replay attacks — if such protection is necessary at all — is left to some higher-level application. The reason the definition of a MAC is structured this way is, once again, because we are unwilling to assume any semantics regarding applications that use MACs; in particular, the decision as to whether or not a replayed message should be treated as "valid" is considered to be entirely application-dependent.

Two common techniques for preventing replay attacks involve the use of *sequence numbers* or *time-stamps*. The basic idea of the first approach is that each message m is assigned a sequence number i, and the MAC tag is computed over the concatenated message $i \| m$. (Actually, it is not quite this simple since the concatenation must be done in such a way that $i \| m$ uniquely determines i and m, but we gloss over such details in this high-level overview.) It is assumed here that the sender always assigns a unique sequence number to each message, and that the receiver keeps track of which sequence numbers it has already seen. Now, any successful replay of a message m will have to forge a valid MAC tag on a new *concatenated* message $i' \| m$, where i' has never been used before. This is ruled out by the security of the MAC.

A disadvantage of using sequence numbers is that the receiver must store a list of all previous sequence numbers it has received. (Though if communication is occurring in a dedicated session, the sender can simply increment the sequence number each time a message is sent, and the receiver need only store the highest sequence number previously received.) To alleviate this, time-stamps are sometimes used to similar effect. Here, the sender essentially appends the current time to the message (say, to the nearest millisecond) rather than a sequence number. When the receiver obtains a message, it checks whether the included time-stamp is within some acceptable window of the current time. This method has certain drawbacks as well, including the need for the sender and receiver to maintain closely-synchronized clocks, and the possibility that a replay attack can still take place as long as it is done quickly enough (specifically, within the acceptable time window).

Further discussion about preventing replay attacks is beyond the scope of this book, but can be found in any good book on Network Security.

4.4 Constructing Secure Message Authentication Codes

Pseudorandom functions are a natural tool for constructing secure message authentication codes. Intuitively, if the MAC tag t is obtained by applying a pseudorandom function to the message m, then forging a tag on a previously-unauthenticated message requires the adversary to guess the value of the pseudorandom function at a "new" point (i.e., message). Now, the probability of guessing the value of a *random* function on a new point is 2^{-n} (when the output length of the function is n). It follows that the probability of guessing such a value for a *pseudorandom* function can be only negligibly greater.

CONSTRUCTION 4.3

Let F be a pseudorandom function. Define a fixed-length MAC for messages of length n as follows:

- Gen: on input 1^n, choose $k \leftarrow \{0,1\}^n$ uniformly at random.
- Mac: on input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^n$, output the tag $t := F_k(m)$. (If $|m| \neq |k|$ then output nothing.)
- Vrfy: on input a key $k \in \{0,1\}^n$, a message $m \in \{0,1\}^n$, and a tag $t \in \{0,1\}^n$, output 1 if and only if $t \stackrel{?}{=} F_k(m)$. (If $|m| \neq |k|$, then output 0.)

A fixed-length MAC from any pseudorandom function.

The above idea, shown in Construction 4.3, works for constructing a secure MAC for *fixed-length* messages. This is already useful, though it falls short of

our ultimate goal. We will see later in this section that any fixed-length MAC can be converted into a MAC that handles messages of arbitrary length.

THEOREM 4.4 *If F is a pseudorandom function, then Construction 4.3 is a fixed-length MAC for messages of length n that is existentially unforgeable under an adaptive chosen-message attack.*

PROOF The intuition behind the proof of this theorem was described above, and so we turn directly to the details. As in previous uses of pseudorandom functions, this proof follows the paradigm of first analyzing the security of the scheme using a truly random function, and then considering the result of replacing the truly random function with a pseudorandom one.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary and define ε as follows:

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1].$$

Consider a message authentication code $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Mac}}, \widetilde{\mathsf{Vrfy}})$ which is the same as $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ in Construction 4.3 except that a truly random function f is used instead of the pseudorandom function F_k. That is, $\widetilde{\mathsf{Gen}}(1^n)$ works by choosing a random function $f \leftarrow \mathsf{Func}_n$, and $\widetilde{\mathsf{Mac}}$ computes a MAC tag just as Mac does except that f is used instead of F_k. (Technically, this is not a legal MAC because it is not efficient. Nevertheless, it is well-defined for the purposes of the proof.) It is straightforward to see that

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \leq 2^{-n} \tag{4.1}$$

because for any message $m \notin \mathcal{Q}$, the value $t = f(m)$ is uniformly distributed in $\{0,1\}^n$ from the point of view of the adversary $\mathcal{A}$.

We now construct a polynomial-time distinguisher D that is given oracle access to some function, and whose goal is to determine whether this function is pseudorandom (i.e., equal to F_k for randomly-chosen $k \leftarrow \{0,1\}^n$) or random (i.e., equal to f for $f \leftarrow \mathsf{Func}_n$). To do this, D emulates the message authentication experiment for $\mathcal{A}$ in the manner described below, and observes whether $\mathcal{A}$ succeeds in outputting a valid tag on a "new" message. If so, D guesses that its oracle must be a pseudorandom function; otherwise, D guesses that its oracle must be a random function. In detail:

> **Distinguisher D:**
> D is given input 1^n and access to an oracle $\mathcal{O} : \{0,1\}^n \to \{0,1\}^n$ and works as follows:
>
> 1. Run $\mathcal{A}(1^n)$. Whenever $\mathcal{A}$ queries its MAC oracle on a message m (i.e., whenever $\mathcal{A}$ requests a tag on a message m), answer this query in the following way:
>   - Query $\mathcal{O}$ with m and obtain response t; return t to $\mathcal{A}$.

2. When $\mathcal{A}$ outputs (m, t) at the end of its execution, do:
   (a) Query $\mathcal{O}$ with m and obtain response $\hat{t}$.
   (b) If **(1)** $\hat{t} = t$ and **(2)** $\mathcal{A}$ never queried its MAC oracle on m, then output 1; otherwise, output 0.

It is clear that D runs in polynomial time.

Notice that if D's oracle is a pseudorandom function, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$. Furthermore, D outputs 1 exactly when $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$. We therefore conclude that

$$\Pr\left[D^{F_k(\cdot)}(1^n) = 1\right] = \Pr\left[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1\right] = \varepsilon(n),$$

where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random in the above. If D's oracle is a random function, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n)$, and again D outputs 1 exactly when $\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1$. Thus,

$$\Pr\left[D^{f(\cdot)}(1^n) = 1\right] = \Pr\left[\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] \leq \frac{1}{2^n},$$

where $f \leftarrow \mathsf{Func}_n$ is chosen uniformly at random in the above. Therefore,

$$\left|\Pr\left[D^{F_k(\cdot)}(1^n) = 1\right] - \Pr\left[D^{f(\cdot)}(1^n) = 1\right]\right| \geq \varepsilon(n) - \frac{1}{2^n}.$$

Since F is a pseudorandom function, it follows that there exists a negligible function negl with $\varepsilon(n) - 2^{-n} \leq \mathsf{negl}(n)$. We then have $\varepsilon(n) \leq \mathsf{negl}(n) + 2^{-n}$, and so ε is negligible. This concludes the proof that Construction 4.3 is existentially unforgeable under an adaptive chosen-message attack. ■

Extension to Variable-Length Messages

Construction 4.3 is important in that it shows a general paradigm for constructing secure message authentication codes based on pseudorandom functions. Unfortunately, the construction is only capable of dealing with *fixed-length* messages that are furthermore rather short. These limitations are unacceptable in many (if not most) applications.[2] We show here how a general (variable-length) MAC can be constructed from any fixed-length MAC for messages of length n. The construction we show is not very efficient and is

[2]Given a pseudorandom function that handles variable-length inputs, Construction 4.3 would yield a secure MAC for messages of arbitrary length. Likewise, a pseudorandom function with a longer, but fixed domain, would yield a secure MAC for longer messages. However, existing *practical* pseudorandom functions (block ciphers) are only defined for short, fixed-length inputs.

unlikely to be used in practice. Indeed, far more efficient constructions of secure variable-length MACs are known and we will discuss these in Sections 4.5 and 4.7. We include this construction for its simplicity and generality.

Let $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ be a secure fixed-length MAC for messages of length n (for simplicity we assume that Gen' chooses a random n-bit key). Before presenting the construction of a variable-length MAC based on Π', we rule out some simple ideas. In all the following (and the secure construction below), the basic idea is to break the message m into blocks $m_1, \ldots, m_d$ and authenticate the blocks using Π' in some way. Consider the following suggestions:

1. *XOR all the blocks together and authenticate the result.* I.e., compute the tag $t := \mathsf{Mac}'_k(\bigoplus_i m_i)$. In this case, an adversary can forge a valid tag on a new message by changing the original message so that the XOR of the blocks does not change. This can easily be done.

2. *Authenticate each block separately.* I.e., compute $t_i := \mathsf{Mac}'_k(m_i)$ and output $\langle t_1, \ldots, t_d \rangle$ as the tag. This prevents an adversary from sending any previously-unauthenticated block without being detected. However, it does not prevent an adversary from changing the order of the blocks, and computing a valid tag on, e.g., the message $m_d, \ldots, m_1$ (something that is not allowed by Definition 4.2).

3. *Authenticate each block along with a sequence number.* I.e., compute $t_i := \mathsf{Mac}'_k(i \| m_i)$ and output $\langle t_1, \ldots, t_d \rangle$ as the tag. This prevents the re-ordering attack described above. However, the adversary is not prevented from dropping blocks from the end of the message (since $\langle t_1, \ldots, t_{d-1} \rangle$ is a valid tag on the message $m_1, \ldots, m_{d-1}$). Furthermore, the adversary can mix-and-match blocks from different messages. That is, if the adversary obtains the tags $\langle t_1, \ldots, t_d \rangle$ and $\langle t'_1, \ldots, t'_d \rangle$ on the messages $m = m_1, \ldots, m_d$ and $m' = m'_1, \ldots, m'_d$, respectively, it can output the valid tag $\langle t_1, t'_2, t_3, t'_4, \ldots \rangle$ on the message $m_1, m'_2, m_3, m'_4, \ldots$

All the attacks described above must be prevented by the eventual solution. This is achieved by including additional information in every block. Specifically, in addition to including a sequence number as above, we also include a random "message identifier" that prevents blocks from different messages from being combined. Finally, each block also includes the length of the message, so that blocks at the end of the message cannot be dropped. The scheme is shown as Construction 4.5.[3]

[3] A technicality is that the construction is only defined for messages of length $\ell < 2^{n/4}$. This is easy to fix by changing Mac so it outputs $\perp$ when $\ell \geq 2^{n/4}$, and changing Vrfy so that it outputs 1 when $\ell \geq 2^{n/4}$. Although this seems "silly", a polynomial-time adversary will be unable to output a message of length $2^{n/4}$ for n large enough, and so security still holds. From a practical point of view we do not expect honest parties to ever send messages of exponential length, and so the original construction suffices.

CONSTRUCTION 4.5

Let $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ be a fixed-length MAC for messages of length n. Define a MAC as follows:

- Gen: this is identical to Gen'.
- Mac: on input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^*$ of length $\ell < 2^{\frac{n}{4}}$, parse m into d blocks $m_1, \ldots, m_d$, each of length $n/4$. (The final block is padded with 0s if necessary.) Next, choose a random identifier $r \leftarrow \{0,1\}^{n/4}$.

  For $i = 1, \ldots, d$, compute $t_i \leftarrow \mathsf{Mac}'_k(r \| \ell \| i \| m_i)$, where i and ℓ are uniquely encoded as strings of length $n/4$.[4] Finally, output the tag $t := \langle r, t_1, \ldots, t_d \rangle$.
- Vrfy: on input a key $k \in \{0,1\}^n$, a message $m \in \{0,1\}^*$ of length $\ell < 2^{\frac{n}{4}}$, and a tag $t = \langle r, t_1, \ldots, t_{d'} \rangle$, parse m into d blocks $m_1, \ldots, m_d$, each of length $n/4$. (The final block is padded with 0s if necessary.) Output 1 if and only if $d' = d$ and $\mathsf{Vrfy}'_k(r \| \ell \| i \| m_i, t_i) = 1$ for $1 \le i \le d$.

A variable-length MAC from any fixed-length MAC.

THEOREM 4.6 *If Π' is a secure fixed-length MAC for messages of length n, then Construction 4.5 is a MAC that is existentially unforgeable under an adaptive chosen-message attack.*

PROOF The intuition is that as long as Π' is secure, an adversary cannot introduce a new block with a valid tag. Furthermore, the extra information included in each block prevents the various attacks (dropping blocks, re-ordering blocks, etc.) sketched earlier. We stress that showing that known attacks are thwarted is far from a proof of security. Rather, we will rigorously prove security (and essentially will show that the above attacks are the "only" possible ones).

Let Π denote the MAC given by Construction 4.5. Let $\mathcal{A}$ be a probabilistic polynomial-time adversary and define $\varepsilon(\cdot)$ as follows:

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1]. \tag{4.2}$$

Let Repeat denote the event that the same message identifier appears in two of the tags returned by the MAC oracle in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$. If $(m, t = \langle r, t_1, \ldots \rangle)$ denotes the final output of $\mathcal{A}$ and ℓ denotes the length of m, let Forge denote the event that at least one of the blocks $r \| \ell \| i \| m_i$ was never previously authenticated by the MAC oracle, yet $\mathsf{Vrfy}'_k(r \| \ell \| i \| m_i, t_i) = 1$. I.e., Forge is the event that $\mathcal{A}$ was able to output a valid tag on a "message" that was not previously authenticated by the fixed-length MAC Π'.

[4] Notice that i and ℓ can be encoded in $n/4$ bits because $i, \ell < 2^{n/4}$.

We have

$$\begin{aligned}\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1] &= \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \mathsf{Repeat}] \\ &+ \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \overline{\mathsf{Forge}}] \\ &+ \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \mathsf{Forge}].\end{aligned} \tag{4.3}$$

We now show that $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \mathsf{Repeat}] \leq \Pr[\mathsf{Repeat}]$ is negligible, and that $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \overline{\mathsf{Forge}}] = 0$.

CLAIM 4.7 *There is a negligible function* negl *with* $\Pr[\mathsf{Repeat}] \leq \mathsf{negl}(n)$.

Let $q(n)$ be the number of MAC oracle queries made by $\mathcal{A}$. To answer the ith oracle query of $\mathcal{A}$, the oracle chooses $r_i \leftarrow \{0,1\}^{n/4}$ uniformly at random. The probability of event Repeat is exactly the probability that $r_i = r_j$ for some $i \neq j$. Applying the "birthday bound" (Lemma A.9), as discussed in Appendix A.4, we have that $\Pr[\mathsf{Repeat}] \leq \frac{q(n)^2}{2^{n/4}}$. (The identifiers are chosen from a set of size $|\{0,1\}^{n/4}| = 2^{n/4}$.) Since $\mathcal{A}$ makes only polynomially-many queries, this value is negligible, thereby proving the claim.

CLAIM 4.8 $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \overline{\mathsf{Forge}}] = 0$.

Say $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$ and Repeat does not occur; we show that this implies Forge occurs. Recalling that (m,t) is the final output of $\mathcal{A}$, let ℓ denote the length of m. Parse m into d blocks $m_1, \ldots, m_d$, each of length $n/4$ (padding at the end with 0s if necessary). View t as $t = \langle r, t_1, \ldots, t_d \rangle$; since $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$, we know that t contains $d+1$ components. We have the following cases:

1. *The identifier r is different from all the identifiers used by the MAC oracle:* This implies that $r\|\ell\|1\|m_1$ was never previously authenticated by the MAC oracle. Because $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$, we know that $\mathsf{Vrfy}'_k(r\|\ell\|1\|m_1, t_1) = 1$. Thus, Forge occurs in this case.

2. *The identifier r was used in exactly one of the MAC tags obtained by $\mathcal{A}$ from its MAC oracle:* Denote by m' the message that $\mathcal{A}$ queried to its oracle for which the reply t' contained the identifier r. Since $m \notin \mathcal{Q}$ we have $m \neq m'$. Let ℓ' be the length of m'. There are two sub-cases here:

   (a) *Case 1: $\ell' \neq \ell$.* Here it is again the case that $r\|\ell\|1\|m_1$ was never previously authenticated by the MAC oracle. (All MAC oracle responses except one used a different identifier $r' \neq r$, and the one oracle response that used the same identifier r used a different length value $\ell' \neq \ell$.) Since $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$, we again know that Forge occurs in this case.

(b) *Case 2:* $\ell' = \ell$. Parse m' into d blocks $m'_1, \ldots, m'_d$ (since $\ell' = \ell$ the number of blocks in m and m' is the same). Because $m' \neq m$, there must exist some i with $m_i \neq m'_i$. But then $r \| \ell \| i \| m_i$ was never previously authenticated by the MAC oracle. (All MAC oracle responses except one used a different identifier $r' \neq r$; the one oracle response that used the same identifier r used a sequence number $i' \neq i$ in all blocks but one, and in the remaining block it used $m'_i \neq m_i$.) Since $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$, we have that Forge occurs in this case as well.

3. *The identifier r was used in more than one of the MAC tags obtained by $\mathcal{A}$ from its oracle:* This cannot occur because Repeat did not occur.

Noting that this covers all possible cases, this completes the proof of the claim, and is really the heart of the proof.

Returning to Equation (4.3) and using the previous two claims, we see that

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \overline{\mathsf{Repeat}} \wedge \mathsf{Forge}] \geq \varepsilon(n) - \mathsf{negl}(n).$$

We now construct a probabilistic polynomial-time adversary $\mathcal{A}'$ who attacks the fixed-length MAC Π' and succeeds in outputting a valid forgery on a previously-unauthenticated message with at least the above probability. The construction of $\mathcal{A}'$ is the obvious one and so we describe it briefly: $\mathcal{A}'$ runs $\mathcal{A}$ as a sub-routine, and answers the request by $\mathcal{A}$ for a MAC tag on the message m by choosing $r \leftarrow \{0,1\}^{n/4}$ itself, parsing m appropriately, and making the appropriate queries to its own MAC oracle. When $\mathcal{A}$ outputs (m, t), with m having length ℓ, adversary $\mathcal{A}'$ parses m as $m_1, \ldots, m_d$ and t as $\langle r, t_1, \ldots, t_{d'} \rangle$ and checks for a previously-unauthenticated block $r \| \ell \| i \| m_i$ (this is easy to do since $\mathcal{A}'$ can keep track of all the queries it makes to its own oracle). If such a block exists, $\mathcal{A}'$ outputs $(r \| \ell \| i \| m_i, t_i)$. If not, $\mathcal{A}'$ outputs nothing

$\mathcal{A}'$ forges a valid MAC tag on a previously-unauthenticated message (all with respect to Π') whenever Forge occurs and so whenever $\mathcal{A}$ forges a valid MAC tag on a previously-unauthenticated message (with respect to Π). It is easy to see that the view of $\mathcal{A}$ when run as a sub-routine by $\mathcal{A}'$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$. So,

$$\begin{aligned}
\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1] &= \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \mathsf{Forge}] \\
&\geq \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \overline{\mathsf{Repeat}} \wedge \mathsf{Forge}] \\
&\geq \varepsilon(n) - \mathsf{negl}(n).
\end{aligned}$$

By security of Π', we know that $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1]$ must be negligible. This implies that ε must be negligible as well. ∎

4.5 CBC-MAC

Theorems 4.4 and 4.6, taken together, show that it is possible to construct a secure message authentication code (for messages of arbitrary length) given only a pseudorandom function that works for a fixed input length. This demonstrates, in principle, that secure MACs can be constructed from block ciphers. Unfortunately, the resulting construction is extremely inefficient: to compute a MAC tag on a message of length $\ell \cdot n$, it is necessary to apply the block cipher 4ℓ times, and the MAC tag is $(4\ell + 1)n$ bits long. Fortunately, it is possible to achieve far more efficient solutions.

The CBC-MAC construction is similar to the CBC mode of encryption and is widely used in practice. As in Construction 4.5, the message is broken up into blocks, and a block cipher is then applied. However, in order to compute a tag on a message of length $\ell \cdot n$, where n is the size of a message block, the block cipher is applied only ℓ times. More importantly, the MAC tag is now only n bits long (i.e., a single block). We begin by presenting the basic CBC-MAC construction which gives a secure *fixed-length* MAC (but for an arbitrary length set in advance). We caution that this basic scheme is *not* secure in the general case where messages of different lengths may be authenticated.

CONSTRUCTION 4.9

Let F be a pseudorandom function, and fix a length function ℓ. The basic CBC-MAC construction is as follows:

- Gen: on input 1^n, choose $k \leftarrow \{0,1\}^n$ uniformly at random.
- Mac: on input a key $k \in \{0,1\}^n$ and a message m of length $\ell(n) \cdot n$, do the following (we set $\ell = \ell(n)$ in what follows):

  1. Parse m as $m = m_1, \ldots, m_\ell$ where each m_i is of length n, and set $t_0 := 0^n$.
  2. For $i = 1$ to ℓ, set $t_i := F_k(t_{i-1} \oplus m_i)$.

  Output t_ℓ as the tag.
- Vrfy: on input a key $k \in \{0,1\}^n$, a message m of length $\ell(n) \cdot n$, and a tag t of length n, output 1 if and only if $t \stackrel{?}{=} \mathsf{Mac}_k(m)$.

CBC-MAC for fixed-length messages.

A graphical depiction of Construction 4.9 (modified to handle variable-length messages, as discussed subsequently) is given in Figure 4.1. The construction is described for messages whose length is a multiple of n, but it is possible to handle messages of arbitrary (but fixed) length using padding. The security of this construction is described by the following theorem:

THEOREM 4.10 *Let ℓ be a polynomial. If F is a pseudorandom function, then Construction 4.9 is a fixed-length MAC for messages of length $\ell(n) \cdot n$ that is existentially unforgeable under an adaptive chosen-message attack.*

The proof of Theorem 4.10 is very involved and is therefore omitted. We stress that even though Construction 4.9 can be extended in the obvious way to handle messages whose length is an arbitrary multiple of n, the construction is only secure when the length of the messages being authenticated is fixed. That is, if an adversary is able to obtain MAC tags for messages of varying lengths, then the scheme is no longer secure (see Exercise 4.8). The advantage of this construction over Construction 4.3, which also gives a fixed-length MAC, is that the present construction can authenticate much longer messages.

CBC-MAC vs. CBC-mode encryption. There are two differences between the basic CBC-MAC and the CBC mode of encryption:

1. CBC-mode encryption uses a *random IV* and this is crucial for obtaining security. In contrast, CBC-MAC uses no IV (or the fixed value $IV = 0^n$) and this is also crucial for obtaining security. Specifically, CBC-MAC using a random IV is not secure.

2. In CBC-mode encryption all blocks t_i (called c_i in the case of CBC-mode encryption) are output by the encryption algorithm as part of the ciphertext, whereas in CBC-MAC only the final block is output. This may seem to be a technical difference resulting from the fact that, for the case of encryption, all blocks must be output in order to enable decryption, whereas for a MAC this is simply not necessary and so is not done. However, if CBC-MAC is modified to output all blocks then it is no longer secure.

In Exercise 4.9 you are asked to verify the above. This is a good illustration of the fact that harmless-looking modifications to cryptographic constructions can render them insecure. It is crucial to always implement a cryptographic construction exactly as specified, and not to introduce any variations (unless you can prove security for the variant scheme). Furthermore, it is crucial to understand the construction. For example, in many cases a cryptographic library provides a programmer with a "CBC function", but it does not distinguish between the use of this function for encryption or for message authentication.

Secure CBC-MAC for variable-length messages. In order to obtain a secure version of CBC-MAC for variable-length messages, Construction 4.9 must be modified. This can be done in a number of ways. Three possible options that can be *proven secure* are:

1. Apply the pseudorandom function (block cipher) to the length ℓ of the input message in order to obtain a key k_ℓ (i.e., set $k_\ell := F_k(\ell)$). Then,

compute the basic CBC-MAC using the key k_ℓ. This ensures that different (and computationally independent) keys are used to authenticate messages of different lengths.

2. *Prepend* the message with its length $|m|$ (encoded as an n-bit string), and then compute the basic CBC-MAC on the resulting message. (This is shown in Figure 4.1.) We stress that appending the block length to the *end* of the message is *not* secure.

3. Change the scheme so that key generation chooses two different keys $k_1 \leftarrow \{0,1\}^n$ and $k_2 \leftarrow \{0,1\}^n$. Then, to authenticate a message m first compute the basic CBC-MAC of m using k_1 and let t be the result; output the tag $\hat{t} := F_{k_2}(t)$.

The third option has the advantage that it is not necessary to know the message length before starting to compute the MAC. Its disadvantage is that it requires two keys. However, at the expense of two additional applications of the pseudorandom function, it is possible to store a single key k and then derive keys $k_1 = F_k(1)$ and $k_2 = F_k(2)$ at the beginning of the computation.

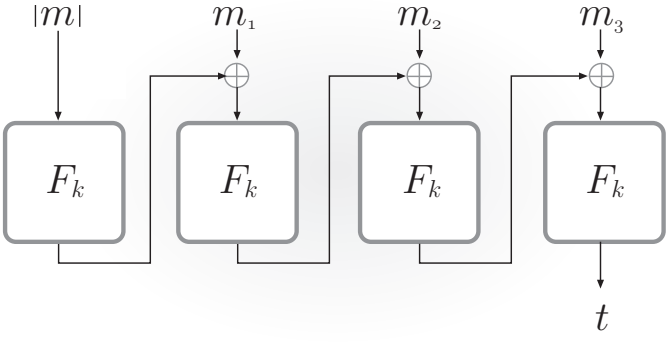

FIGURE 4.1: A variant of CBC-MAC secure for authenticating variable-length messages.

4.6 Collision-Resistant Hash Functions

In this section we take a brief detour from our discussion of message authentication in order to introduce the notion of *collision-resistant hash functions*. Such functions are very useful throughout cryptography, and are introduced now because we will use them in the next section to obtain another efficient construction of a MAC for variable-length messages. (Those who will not cover Section 4.7 can defer the current section until they reach Chapter 12.)

In general, hash functions are just functions that take arbitrary-length strings and *compress* them into shorter strings. The classic use of hash functions is in data structures, where they can be used to achieve $\mathcal{O}(1)$ insertion and lookup times for storing a set of elements. Specifically, if the range of the hash function H is of size N, a table of length N is initialized. Then, element x is stored in cell $H(x)$ of the table. In order to retrieve x, it suffices to compute $H(x)$ and probe that cell of the table for whatever elements are stored there. A "good" hash function for this purpose is one that yields as few *collisions* as possible, where a collision is a pair of distinct data items x and x' for which $H(x) = H(x')$. Notice that when a collision occurs, two elements end up being stored in the same cell. Therefore, many collisions result in a higher-than-desired retrieval complexity. In short, what is desired is that the hash function spreads the elements well in the table, thereby minimizing the number of collisions.

Collision-resistant hash functions are similar in principle to those used in data structures. In particular, they are also functions that compress arbitrary-length input strings into output strings of some fixed length. As in data structures, the goal is to avoid collisions. However, there are fundamental differences between standard hash functions and collision-resistant ones. For one, the *desire* to minimize collisions in the setting of data structures becomes a *mandatory requirement* to avoid collisions in the setting of cryptography. Furthermore, in the context of data structures we can assume that the set of data elements is chosen independently of the hash function and without any intention to cause collisions. In the context of cryptography, in contrast, we need to be concerned with an adversary who may select elements depending on the hash function with the explicit goal of causing collisions. This means that the requirements on hash functions used in cryptography are much more stringent than the analogous requirements in data structures, and collision-resistant hash functions are therefore much harder to construct.

4.6.1 Defining Collision Resistance

A *collision* in a function H is a pair of distinct inputs x and x' such that $H(x) = H(x')$; in this case we also say that x and x' *collide* under H. A function H is *collision resistant* if it is infeasible for any probabilistic polynomial-time algorithm to find a collision in H. Typically we will be interested in functions H that have an infinite domain (e.g., they take as input strings of all possible lengths) and a finite range. In such a case, collisions must *exist* by the pigeon-hole principle, and the requirement is therefore only that such collisions should be "hard" to find. We will sometimes refer to functions H for which both the input domain and output range are finite. In this case, however, we will only be interested in functions that compress their input, meaning that the length of the output is shorter than that of the input. Collision resistance is trivial to achieve if compression is not required: for example, the identity function is trivially collision resistant.

Formally, we will deal with a *family* of hash functions indexed by a "key" s. That is, H will be a two-input function that takes as inputs a key s and a string x, and outputs a string $H^s(x) \stackrel{\text{def}}{=} H(s,x)$. The requirement is that it must be hard to find a collision in H^s for a randomly-generated s. The key s is not a usual cryptographic key, and there are at least two differences from our treatment of keyed functions in the context of pseudorandom functions. First, not all strings necessarily correspond to valid keys (i.e., H^s may not be defined for certain s), and therefore the key s will be generated by an algorithm Gen rather than being chosen uniformly at random. Second, and perhaps more important, this key s is not kept secret. Rather, it is used merely to specify a particular function H^s from the family. In order to emphasize the fact that s is not kept secret, we superscript the key and write H^s rather than H_s.

DEFINITION 4.11 *A* hash function *is a pair of probabilistic polynomial-time algorithms* (Gen, H) *satisfying the following:*

- Gen *is a probabilistic algorithm which takes as input a security parameter* 1^n *and outputs a key* s*. We assume that* 1^n *is implicit in* s*.*
- *There exists a polynomial* ℓ *such that* H *takes as input a key* s *and a string* $x \in \{0,1\}^*$ *and outputs a string* $H^s(x) \in \{0,1\}^{\ell(n)}$ *(where* n *is the value of the security parameter implicit in* s*).*

If H^s *is defined only for inputs* $x \in \{0,1\}^{\ell'(n)}$ *and* $\ell'(n) > \ell(n)$*, then we say that* (Gen, H) *is a* fixed-length *hash function for inputs of length* $\ell'(n)$*.*

In the fixed-length case we require that ℓ' be greater than ℓ. This ensures that the function is a hash function in the classic sense in that it *compresses* its input. In the general case we have no requirement on ℓ because the function takes as input all (finite) binary strings, and so in particular strings that are longer than $\ell(n)$. Thus, by definition, it also compresses (albeit only strings that are longer than $\ell(n)$).

We now proceed to define security. As usual, we first define an experiment for a hash function $\Pi = (\mathsf{Gen}, H)$, an adversary $\mathcal{A}$, and a security parameter n:

The collision-finding experiment $\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n)$**:**

1. *A key* s *is generated by running* $\mathsf{Gen}(1^n)$*.*
2. *The adversary* $\mathcal{A}$ *is given* s *and outputs* x, x'*. (If* Π *is a fixed-length hash function for inputs of length* $\ell'(n)$ *then we require* $x, x' \in \{0,1\}^{\ell'(n)}$*.)*
3. *The output of the experiment is defined to be 1 if and only if* $x \neq x'$ *and* $H^s(x) = H^s(x')$*. In such a case we say that* $\mathcal{A}$ *has found a collision.*

The definition of collision resistance states that no efficient adversary can find a collision in the above experiment except with negligible probability.

DEFINITION 4.12 *A hash function* $\Pi = (\mathsf{Gen}, H)$ *is* collision resistant *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n) = 1\right] \leq \mathsf{negl}(n).$$

Terminology: For simplicity, we refer to H, H^s, and $\Pi = (\mathsf{Gen}, H)$ as "collision-resistant hash functions." This should not cause any confusion.

4.6.2 Weaker Notions of Security for Hash Functions

Collision resistance is a strong security requirement and is quite difficult to achieve. However, in some applications it suffices to rely on more relaxed requirements. When considering cryptographic hash functions, there are typically three levels of security considered:

1. *Collision resistance:* This is the strongest notion and the one we have considered so far.

2. *Second pre-image resistance:* Informally speaking, a hash function is second pre-image resistant if given s and x it is infeasible for a probabilistic polynomial-time adversary to find $x' \neq x$ such that $H^s(x') = H^s(x)$.

3. *Pre-image resistance:* Informally, a hash function is pre-image resistant if given s and $y = H^s(x)$ (but not x itself) for a randomly-chosen x, it is infeasible for a probabilistic polynomial-time adversary to find a value x' such that $H^s(x') = y$. (Looking ahead to later chapters of the book, this essentially means that H^s is *one-way*.)

We do not formally define the latter two notions since they will not be used in this book, but leave this task as an exercise.

Any hash function that is collision resistant is also second pre-image resistant. Intuitively, this is the case because if given x an adversary can find $x' \neq x$ for which $H^s(x') = H^s(x)$, then it can clearly find a colliding pair x and x' from scratch. Likewise, any hash function that is second pre-image resistant is also pre-image resistant. This is due to the fact that if it were possible to invert y and find an x' such that $H^s(x') = y$, then it would be possible to take a given input x, compute $y := H^s(x)$, and then invert y to obtain x'; with high probability $x' \neq x$ (relying on the fact that H compresses, and so many different inputs map to the same output) in which case a second pre-image has been found. (You are asked to formalize the above arguments in Exercise 4.10.) We conclude that the above three security requirements form a hierarchy with each definition implying the one below it.

4.6.3 A Generic "Birthday" Attack

Before we show how to construct collision-resistant hash functions, we present a generic attack that finds collisions in *any* hash function (albeit in time that is exponential in the hash output length). This attack implies a minimal output length needed for a hash function to potentially be secure against adversaries running for a certain time, as we will explain.

Assume we are given a hash function $H : \{0,1\}^* \to \{0,1\}^\ell$. (For simplicity we deal with hash functions taking arbitrary-length inputs, though a slight variant of the attack works for fixed-length hash functions also. We omit the key s since the attack here applies independently of any key.) The attack works as follows: Choose q arbitrary distinct inputs $x_1, \ldots, x_q \in \{0,1\}^{2\ell}$, compute $y_i := H(x_i)$, and check whether any of the two y_i values are equal.

What is the probability that this algorithm finds a collision? Clearly, if $q > 2^\ell$, this occurs with probability 1. However, we are interested in the case of smaller q. It is somewhat difficult to analyze this probability exactly, and so we will instead analyze an idealized case in which H is treated as a random function.[5] That is, for each i we assume that the value $y_i = H(x_i)$ is uniformly distributed in $\{0,1\}^\ell$ and independent of any of the previous output values $\{y_j\}_{j<i}$ (recall we assume all $\{x_i\}$ are distinct). We have thus reduced our problem to the following one: if we choose values $y_1, \ldots, y_q \leftarrow \{0,1\}^\ell$ uniformly at random, what is the probability that there exist distinct i, j with $y_i = y_j$?

This problem has been extensively studied, and is related to the so-called *birthday problem* discussed in detail in Appendix A.4. For this reason, the collision-finding algorithm we have described is often called a "birthday attack." The birthday problem is the following: if q people are in a room, what is the probability that two of them have the same birthday? (Assume birthdays are uniformly and independently distributed among the 365 days of a non-leap year.) This is exactly analogous to our problem: if y_i represents the birthday of person i, then we have $y_1, \ldots, y_q \leftarrow \{1, \ldots, 365\}$ chosen uniformly at random. Furthermore, matching birthdays correspond to distinct i, j with $y_i = y_j$ (i.e., matching birthdays correspond to collisions).

When $q = \Theta(2^{\ell/2})$, the probability of such a collision is roughly $1/2$. In the case of birthdays, it turns out that if there are only 23 people in the room, the probability that two have the same birthday is greater than $1/2$. This is proven formally in Appendix A.4.

Birthday attacks on hash functions — summary. If the output length of a hash function is ℓ bits then the birthday attack finds a collision with high probability using $\mathcal{O}(q) = \mathcal{O}(2^{\ell/2})$ hash-function evaluations (by sorting the outputs, a collision can be found — if one exists — in time $\mathcal{O}(\ell \cdot 2^{\ell/2})$;

[5] Actually, it can be shown that this is (essentially) the worst case, and the algorithm finds collisions with higher probability if H deviates significantly from random.

for simplicity, we assume that evaluating H can be done in constant time). We therefore conclude that for the hash function to resist collision-finding attacks that run in time T, the output length of the hash function needs to be at least $2 \log T$ bits. When considering asymptotic bounds on security, there is no difference between a naive attack that tries $2^\ell + 1$ elements and a birthday attack that tries $2^{\ell/2}$ elements: if $\ell(n) = \mathcal{O}(\log n)$ then both attacks run in polynomial time, but if $\ell(n)$ is super-logarithmic then both attacks do not. Nevertheless, in practice birthday attacks make a huge difference. As an example, assume a hash function is designed with output length of 128 bits. It is clearly infeasible to run 2^{128} steps in order to find a collision. However, running for 2^{64} steps is within the realm of feasibility (though still rather difficult). Thus, the existence of generic birthday attacks mandates that any collision-resistant hash function in practice needs to have output that is longer than 128 bits. We stress that having a long enough output is only a *necessary* condition for meeting Definition 4.12, but is very far from being a sufficient one. We also stress that birthday attacks work only for collision resistance; there are no generic attacks on hash functions for second pre-image or pre-image resistance that run in time better than 2^ℓ.

Improved birthday attacks. The birthday attack described above has two weaknesses. First, it requires a large amount of memory. Second, the attack gives very little control over the colliding values. It is possible to construct better birthday attacks that avoid these weaknesses.

The basic birthday attack requires the attacker to store all q values $\{y_i\}$, because the attacker does not know in advance which pair of values will yield a collision. This is a significant drawback because memory is, in general, a scarcer resource than time. For an illustrative (but completely ad-hoc) example, compare 2^{60} bytes to 2^{60} CPU instructions: 2^{60} bytes is about 1 billion gigabytes. Using the largest commercially-available storage devices — that hold roughly 1,000 gigabytes — means that 1 million such devices would be needed. In contrast, 2^{60} instructions can be carried out in about 2 years (assuming a CPU carrying out 25 billion instructions per second, which represents the high end of currently-available personal computers). While this is a long time to wait, it certainly represents a feasible computation. Furthermore, computations of this complexity have actually been carried out before using large distributed networks.

We see that a birthday-type attack becomes much more feasible if its space requirements can be decreased. In fact, it is possible to carry out a birthday-type attack with similar time complexity and success probability as before but using only a *constant* amount of memory. The idea is to take a random initial value x_0 and then, for $i > 0$, compute the values $x_i := H(x_{i-1})$ and $x_{2i} := H(H(x_{2(i-1)}))$. In each step the values x_i and x_{2i} are compared; if they are equal then x_{i-1} and $H(x_{2(i-1)})$ are a collision (unless they happen to be equal, which occurs with negligible probability if we continue to model H as a random function). The key point is that the memory required here

is only that needed to store the values x_i and x_{2i}. This approach can be shown to give a collision with probability roughly 1/2 in $q = \Theta(2^{\ell/2})$ steps; see Section 8.1.2 for analysis of a similar idea used in a different context.

The second weakness that we mentioned relates to the lack of control over the colliding messages that are found. Although it is not necessary to find "meaningful" collisions in order to violate the formal definition of collision resistance, it is still nice to see that birthday attacks can find "meaningful" collisions too. Assume an attacker Alice wishes to find two messages x and x' such that $H(x) = H(x')$, and furthermore the first message x should be a letter from her employer explaining why she was fired from work, while the second message x' should be a flattering letter of recommendation. The birthday attack only relies on the fact that the messages $x_1, \ldots, x_q$ are distinct, but these messages do not need to be random. Thus, we can carry out a birthday-type attack by generating $q = \Theta(2^{\ell/2})$ messages of the first type and q messages of the second type, and then looking for collisions between messages of the two types. It may seem unlikely that this can be done for the aforementioned letters; however, a little thought shows that it is easy to write the same sentence in many different ways. For example, consider the following:

> It is *hard/difficult/challenging/impossible* to *imagine/believe* that we will *find/locate/hire* another *employee/person* having similar *abilities/skills/character* as Alice. She has done a *great/super* job.

The point to notice is that any combination of the italicized words is possible. Thus, the sentence can be written in $4 \cdot 2 \cdot 3 \cdot 2 \cdot 3 \cdot 2 = 288$ different ways. This is just one sentence and so it is actually easy to write a letter that can be rewritten in 2^{64} different ways (you just need 64 words with one synonym each). Using this idea it is possible for the attacker to prepare $2^{\ell/2}$ letters explaining why the attacker was fired and another $2^{\ell/2}$ letters of recommendation; with good probability, a collision between the two types of letters will be found. This attack does require a large amount of memory and the low-memory version described above cannot be used here.

4.6.4 The Merkle-Damgård Transform

We now present an important methodology called the *Merkle-Damgård transform* that is widely used for constructing collision-resistant hash functions in practice. The methodology enables a conversion from any fixed-length hash function to a full-fledged hash function (i.e., one handling inputs of arbitrary length) while maintaining the collision resistance property (if present) of the former. This means that when designing collision-resistant hash functions, we can restrict our attention to the fixed-length case. This in turn makes the job of designing practical collision-resistant hash functions much easier. In addition to being used extensively in practice, the Merkle-Damgård transform is

interesting from a theoretical point of view since it implies that compressing by a single bit is as easy (or as hard) as compressing by an arbitrary amount.

For concreteness, we consider the case that we are given a fixed-length collision-resistant hash function that compresses its input by half; that is, the input length is $\ell'(n) = 2\ell(n)$ and the output length is $\ell(n)$. In Exercise 4.14 you are asked to generalize the construction for any $\ell' > \ell$. We denote the fixed-length collision-resistant hash function by (Gen, h) and use it to construct a collision-resistant hash function (Gen, H) that maps inputs of any length to outputs of length $\ell(n)$. (Gen will remain unchanged.) In much of the literature, the fixed-length collision-resistant hash function is called a *compression function*. The Merkle-Damgård transform is defined in Construction 4.13 and depicted in Figure 4.2.

CONSTRUCTION 4.13

Let (Gen, h) be a fixed-length collision-resistant hash function for inputs of length $2\ell(n)$ and with output length $\ell(n)$. Construct a variable-length hash function (Gen, H) as follows:

- Gen: remains unchanged.
- H: on input a key s and a string $x \in \{0,1\}^*$ of length $L < 2^{\ell(n)}$, do the following (set $\ell = \ell(n)$ in what follows):
  1. Set $B := \lceil \frac{L}{\ell} \rceil$ (i.e., the number of blocks in x). Pad x with zeroes so its length is a multiple of ℓ. Parse the padded result as the sequence of ℓ-bit blocks $x_1, \ldots, x_B$. Set $x_{B+1} := L$, where L is encoded using exactly ℓ bits.
  2. Set $z_0 := 0^\ell$.
  3. For $i = 1, \ldots, B+1$, compute $z_i := h^s(z_{i-1} \| x_i)$.
  4. Output z_{B+1}.

The Merkle-Damgård transform.

We limit the length of x to be at most $2^{\ell(n)} - 1$ so that its length L can be encoded as an integer of length $\ell(n)$. See footnote 3 for an explanation as to why this is not a limitation in either a practical or a theoretical sense.

The initialization vector. The value z_0 used in step 2 of Construction 4.13 is arbitrary and can be replaced by any constant. This value is typically called the IV or *initialization vector*.

The security of the Merkle-Damgård transform. The intuition behind the security of the Merkle-Damgård transform is that if two different strings x and x' collide in H^s, then there must be *distinct* intermediate values $z_{i-1} \| x_i$ and $z'_{i-1} \| x'_i$ in the computation of $H^s(x)$ and $H^s(x')$, respectively, such that $h^s(z_{i-1} \| x_i) = h^s(z'_{i-1} \| x'_i)$. Stated differently, a collision in H^s can only occur

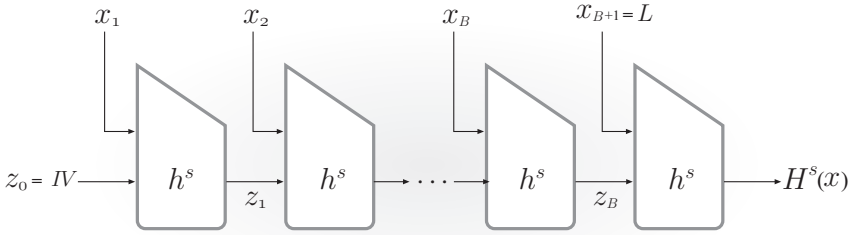

FIGURE 4.2: The Merkle-Damgård transform.

if there is a collision in the underlying h^s. We demonstrate this in the proof by showing that if such a collision in h^s does not occur, then x must equal x' (in contradiction to the assumption that x and x' constitute a collision in H^s). We now proceed to the formal proof.

THEOREM 4.14 *If* (Gen, h) *is a fixed-length collision-resistant hash function, then* (Gen, H) *is a collision-resistant hash function.*

PROOF We show that for any s, a collision in H^s yields a collision in h^s. Let x and x' be two different strings of respective lengths L and L' such that $H^s(x) = H^s(x')$. Let $x_1, \ldots, x_B$ be the B blocks of the padded x, and let $x'_1, \ldots, x'_{B'}$ be the B' blocks of the padded x'. Recall that $x_{B+1} = L$ and $x'_{B'+1} = L'$. There are two cases to consider:

1. *Case 1:* $L \neq L'$. In this case, the last step of the computation of $H^s(x)$ is $z_{B+1} := h^s(z_B \| L)$ and the last step of the computation of $H^s(x')$ is $z'_{B'+1} := h^s(z'_{B'} \| L')$. Since $H^s(x) = H^s(x')$ it follows that $h^s(z_B \| L) = h^s(z'_{B'} \| L')$. However, $L \neq L'$ and so $z_B \| L$ and $z'_{B'} \| L'$ are two different strings that collide for h^s.

2. *Case 2:* $L = L'$. Note this means that $B = B'$ and $x_{B+1} = x'_{B+1}$. Let z_i and z'_i be the intermediate hash values of x and x' during the computation of $H^s(x)$ and $H^s(x')$, respectively. Since $x \neq x'$ but $|x| = |x'|$, there must exist at least one index i (with $1 \leq i \leq B$) such that $x_i \neq x'_i$. Let $i^* \leq B + 1$ be the *highest* index for which it holds that $z_{i^*-1} \| x_{i^*} \neq z'_{i^*-1} \| x'_{i^*}$. If $i^* = B + 1$ then $z_B \| x_{B+1}$ and $z'_B \| x'_{B+1}$ are two different strings that collide for h^s because

$$h^s(z_B \| x_{B+1}) = z_{B+1} = H^s(x) = H^s(x') = z'_{B+1} = h^s(z'_B \| x'_{B+1}).$$

   If $i^* \leq B$, then maximality of i^* implies $z_{i^*} = z'_{i^*}$. Thus, once again, $z_{i^*-1} \| x_{i^*}$ and $z'_{i^*-1} \| x'_{i^*}$ are two different strings that collide for h^s.

It follows that any collision in the hash function H^s yields a collision in the fixed-length hash function h^s. It is straightforward to turn this into a formal security reduction, and we leave this for an exercise. ■

4.6.5 Collision-Resistant Hash Functions in Practice

As in the case of pseudorandom functions/permutations, constructions of collision-resistant hash functions come in two forms: provably-secure constructions based on certain number-theoretic assumptions[6] or highly-efficient constructions that are more heuristic in nature. We shall see a construction of the former type in Section 7.4.2. For now we turn our attention to the latter type, which includes those hash functions that are used exclusively in practice.

One important difference between collision-resistant hash functions used in practice and the notion as we have presented it here is that hash functions used in practice are generally *unkeyed*. This means that a *fixed* hash function H is defined, and there is no longer any notion of a Gen algorithm generating a key for H (and, indeed, H takes no key). From a purely theoretical point of view, it is necessary to include keys in any discussion of collision-resistant hash functions since it is difficult to define a meaningful notion of collision resistance for unkeyed hash functions.[7] The most that can be claimed about an unkeyed hash function is that it is infeasible to come up with an algorithm that runs in some "reasonable" amount of time (say, 75 years) and finds a collision in H. In practice, this sort of security guarantee is enough.

Even on a pragmatic level, keyed hash functions have advantages: for one, if a collision is ever found in an unkeyed function H (say, by mounting an exhaustive search taking many years) then H is no longer collision resistant in any meaningful sense, and must be replaced. If H were a keyed function then a collision for H^s that was found using brute-force search does not necessarily make it any easier to find a collision in $H^{s'}$ for a freshly-generated key s'; thus, H can continue to be used as long as the key is updated.

Although we cannot hope to prove collision resistance for the hash functions used in practice (in particular, because they are unkeyed), this does not mean that we do away with proofs altogether when using such hash functions within some larger construction. Proofs of security that rely on collision resistance all show that if the construction under consideration can be "broken" by some polynomial-time adversary, then a collision can be found in the underlying hash function in polynomial time (we have seen one such example when we proved security of the Merkle-Damgård transform). When considering unkeyed functions, this could be translated into a statement of the following form: "if an adversary breaks the given construction in some amount of time,

[6] Interestingly, the assumptions currently needed to construct collision-resistant hash functions are stronger than those needed to construct pseudorandom permutations, and there is some indication that this might be inherent.

[7] To get a sense for the technical problem, let x, x' be a collision for the fixed hash function H (if H handles inputs longer than its output length then such x, x' surely exist). Now, consider the (constant-time) algorithm that simply outputs x and x'. Such an algorithm finds a collision in H with probability 1. Note that an analogous algorithm that outputs a collision in H^s for a *randomly-chosen* (rather than fixed) key s does *not* exist.

then it is possible to find a collision in the hash function in a similar[8] amount of time". If we believe that it is hard to find a collision in the given hash function in any reasonable amount of time, then this gives a reasonable security guarantee for the larger construction.

Coming back to practical constructions, the "birthday attack" discussed previously gives a lower bound on the output length of a hash function that is required in order to achieve some level of security: if the hash function should be collision resistant against adversaries running in time 2^ℓ, then the output length of the hash function should be at least 2ℓ bits. Good collision-resistant hash functions in practice have an output length of at least 160 bits, meaning that a birthday attack would take time 2^{80}, something out of reach today.

Two popular hash functions are MD5 and SHA-1. (As discussed below, due to recent attacks MD5 is no longer secure and should not be used in any application requiring collision resistance. We include it here because MD5 is still used in legacy code.) Both MD5 and SHA-1 first define a *compression function* that compresses fixed-length inputs by a relatively small amount (in our terms, this compression function is a *fixed-length* collision-resistant hash function). Then the Merkle-Damgård transform (or something very similar) is applied to the compression function in order to obtain a collision-resistant hash function for arbitrary-length inputs. The output length of MD5 is 128 bits and that of SHA-1 is 160 bits. The longer output length of SHA-1 makes the generic "birthday attack" more difficult: for MD5, a birthday attack requires $\approx 2^{128/2} = 2^{64}$ hash computations, while for SHA-1 such an attack requires $\approx 2^{160/2} = 2^{80}$ hash computations.

In 2004, a team of Chinese cryptanalysts presented a breakthrough attack on MD5 and a number of related hash functions. Their technique for finding collisions gives little control over the collisions that are found; nevertheless, it was later shown that their method (and in fact any method that finds "random collisions") can be used to find collisions between, for example, two postscript files generating whatever viewable content is desired. A year later, the Chinese team showed (theoretical) attacks on SHA-1 that would find collisions using less time than that required by a generic birthday attack. The attack on SHA-1 requires time 2^{69} which lies outside the current range of feasibility; as of yet, no explicit collision in SHA-1 has been found. (This is in contrast to the attack on MD5, which finds collisions in minutes.)

These attacks have motivated a shift toward stronger hash functions with larger outputs lengths which are less susceptible to the known set of attacks on MD5 and SHA-1. Notable in this regard is the SHA-2 family, which extends SHA-1 and includes hash functions with 256- and 512-bit output lengths. Another ramification of the attacks is that there is now great interest in designing new hash functions and developing a new hash standard.

[8] Actually, this is not always the case: in some proofs, an adversary running for some time t that "breaks" the construction is translated into an adversary running for *much longer time* (say, t^2) that finds a collision. In this case the security guarantee may not be very useful.

4.7 * NMAC and HMAC

Until now we have seen constructions of message authentication codes that are based on pseudorandom functions (or block ciphers). A different approach is taken in the NMAC and HMAC constructions which are based on collision-resistant hash functions constructed using the Merkle-Damgård transform applied to some underlying compression function (where a compression function is the popular term in this context for a fixed-length collision-resistant hash function). Most known collision-resistant hash functions are constructed in this way and so NMAC and HMAC have wide applicability. Loosely speaking, the security of NMAC and HMAC relies on the assumption that — in addition to collision resistance — the compression function has certain pseudorandom properties. This assumption is believed to be true of the compression functions used in most practical hash functions. Below, we discuss more precisely the assumptions required of the compression function.

In the next section we describe NMAC and briefly discuss its security. We then present HMAC, which can be viewed as a special case of NMAC. As usual, our descriptions of the constructions are not complete. In particular, we do not specify how the input messages are padded before the computation begins.

Notation — the IV in Merkle-Damgård. In this section we will explicitly refer to the IV used in the Merkle-Damgård transform of Construction 4.13; recall that this is the (fixed) value assigned to z_0. In the standard Merkle-Damgård construction, the IV is fixed to an arbitrary constant; here, however, we will wish to vary it. We denote by $H^s_{IV}(x)$ the computation of Construction 4.13 on input x with key s, and with z_0 set to the value $IV \in \{0,1\}^\ell$.

4.7.1 Nested MAC (NMAC)

Let H be a hash function constructed using the Merkle-Damgård transform applied to the compression function h. For simplicity, we let n denote the output length of H and h (rather than $\ell(n)$ as before), and we continue to assume that h compresses its input by half. The first step in the construction of NMAC is to define *secretly-keyed* versions of the compression and hash functions. Let s be a fixed, non-secret key and define a secretly-keyed version of h^s by $h^s_k(x) \stackrel{\text{def}}{=} h^s(k\|x)$. That is, the secretly-keyed compression function works by applying the unkeyed compression function to the concatenation of the secret key k and the message. Define H^s_k to be the hash function obtained by setting $IV = k$ in the Merkle-Damgård construction. (This is consistent with the notation introduced above for $H^s_{IV}(x)$.) Note that in the first iteration of the Merkle-Damgård transform, the value $z_1 := h^s(IV\|x_1)$ is computed. Here $IV = k$ and so we obtain that $z_1 = h^s(k\|x_1) = h^s_k(x_1)$.

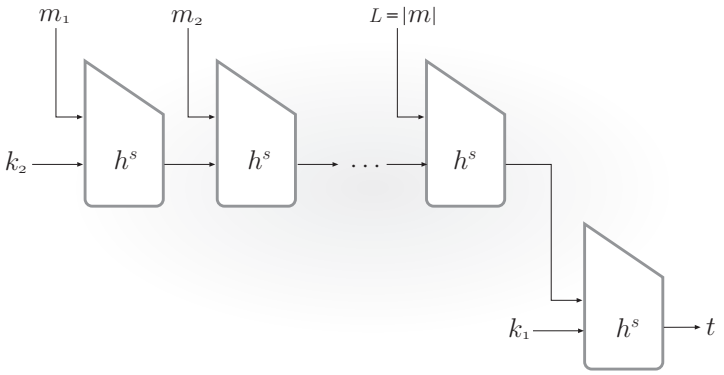

FIGURE 4.3: NMAC.

We are now ready to define NMAC. In words, NMAC works by first applying a secretly-keyed collision-resistant hash function $H^s_{k_2}$ to the input m, and then applying a secretly-keyed compression function $h^s_{k_1}$ to the result. (See Construction 4.15 and Figure 4.3.) $H^s_{k_2}$ is called the *inner* function and $h^s_{k_1}$ is called the *outer* function.

CONSTRUCTION 4.15

Let $(\widetilde{\mathsf{Gen}}, h)$ be a fixed-length collision-resistant hash function, and let $(\widetilde{\mathsf{Gen}}, H)$ be the result of applying the Merkle-Damgård transform to $(\widetilde{\mathsf{Gen}}, h)$. NMAC defines a MAC as follows:

- Gen: on input 1^n, run $\widetilde{\mathsf{Gen}}(1^n)$ to obtain a key s. Also choose $k_1, k_2 \leftarrow \{0,1\}^n$ at random. Output the key (s, k_1, k_2).
- Mac: on input a key (s, k_1, k_2) and a message $m \in \{0,1\}^*$, output the tag $t := h^s_{k_1}(H^s_{k_2}(m))$.
- Vrfy: on input a key (s, k_1, k_2), a message $m \in \{0,1\}^*$, and a tag t, output 1 if and only if $t \stackrel{?}{=} \mathsf{Mac}_{s,k_1,k_2}(m)$.

Nested MAC (NMAC).

The security of NMAC relies on the assumption that h^s_k with key k constitutes a secure MAC. Formally, given a fixed-length hash function $(\widetilde{\mathsf{Gen}}, h)$, define the fixed-length message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ for messages of length n as follows: $\mathsf{Gen}(1^n)$ runs $\widetilde{\mathsf{Gen}}(1^n)$ to obtain s and also picks a key $k \leftarrow \{0,1\}^n$ uniformly at random; the key is (s, k). Then set $\mathsf{Mac}_{s,k}(m) \stackrel{\text{def}}{=} h^s_k(m)$ for $m \in \{0,1\}^n$, and define Vrfy in the natural way. We say that $(\widetilde{\mathsf{Gen}}, h)$ *yields a secure MAC* if Π defined in this way is a secure fixed-length MAC.

THEOREM 4.16 *Let* $(\widetilde{\mathsf{Gen}}, H)$ *denote the Merkle-Damgård transform applied to* $(\widetilde{\mathsf{Gen}}, h)$*. If* $(\widetilde{\mathsf{Gen}}, h)$ *is collision resistant and yields a secure MAC (as defined above), then NMAC is existentially unforgeable under an adaptive chosen-message attack (for arbitrary-length messages).*

Essentially, the NMAC construction first hashes the message m to be authenticated (using a collision-resistant hash function) and then applies a fixed-length message authentication code to the result (where this MAC is also based on the same hash function). It is analogous to the "hash-and-sign" approach discussed extensively in Section 12.4, and we therefore content ourselves here with merely sketching the ideas behind the proof of security for NMAC. Assume toward a contradiction that there exists a probabilistic polynomial-time adversary $\mathcal{A}$ attacking NMAC that forges a valid tag on a new message with non-negligible probability. Recall that $\mathcal{A}$ is given a MAC oracle which it can query for a tag on any messages of its choice. Let m^* denote the message for which $\mathcal{A}$ produces its forgery, and let $\mathcal{Q}$ denote the set of queries made by $\mathcal{A}$ to its MAC oracle (i.e., the set of messages for which it obtained a MAC tag). Assume without loss of generality that $m^* \notin \mathcal{Q}$ (since $\mathcal{A}$ cannot succeed otherwise). There are two possible cases:

1. *Case 1: there exists a message* $m \in \mathcal{Q}$ *such that* $H^s_{k_2}(m^*) = H^s_{k_2}(m)$. In this case, the MAC tag for m is equal to the MAC tag for m^* and so clearly $\mathcal{A}$ can successfully forge a valid tag on m^*. However, this case directly contradicts the assumption that H is collision resistant because $\mathcal{A}$ has found distinct m and m^* for which $H^s_{k_2}(m^*) = H^s_{k_2}(m)$. (By Theorem 4.14, collision resistance of $(\widetilde{\mathsf{Gen}}, h)$ implies collision resistance of $(\widetilde{\mathsf{Gen}}, H_{k_2})$ for *any* value of k_2.)
2. *Case 2: for every message* $m \in \mathcal{Q}$ *it holds that* $H^s_{k_2}(m^*) \neq H^s_{k_2}(m)$. Define $\mathcal{Q}' = \{H^s_{k_2}(m) \mid m \in \mathcal{Q}\}$. The important observation here is that m^* is such that $H^s_{k_2}(m^*) \notin \mathcal{Q}'$. In this case, then, $\mathcal{A}$ is forging a valid tag on the "new message" $H^s_{k_2}(m^*)$ with respect to the fixed-length message authentication code Π described immediately prior to Theorem 4.18. This contradicts the assumption that Π is a secure MAC.

For a formal proof of the above, we refer the reader to Theorem 12.5, where an almost identical proof is given in the context of digital signature schemes. It is straightforward to translate that proof into one that works here as well.

Security assumptions. We caution that the assumption that $(\widetilde{\mathsf{Gen}}, h)$ gives a secure MAC is *not* implied by the assumption that it is collision resistant. Nevertheless, existing practical compression functions (like the one used in SHA-1) are assumed to satisfy this additional requirement.

The key s in Construction 4.15 does not need to be secret in order for security of NMAC to hold. This means that a system-wide key s can be chosen once-and-for-all and used by all parties, and indeed this is what is done in practice when using an unkeyed hash function (which can be viewed as a hash function using a fixed key s).

The key k_2 in Construction 4.15 is not needed once we assume that $(\widetilde{\mathsf{Gen}}, h)$ is collision resistant. (That is, we could have simply fixed $k_2 = 0^n$ as in our original description of the Merkle-Damgård transform.) Indeed, we did not rely on k_2 anywhere in the proof sketch given above. The reason for the introduction of k_2 in NMAC is that it allows a proof of security for NMAC based on a potentially weaker assumption. Specifically, consider the following modified definition of collision resistance: a key s is generated using $\widetilde{\mathsf{Gen}}$ and a random $k_2 \leftarrow \{0,1\}^n$ is also chosen. Then the adversary is allowed to interact with a "hash oracle" that returns $H^s_{k_2}(x)$ in response to the query x. (A variant would be to also give the adversary the non-secret key s.) The adversary succeeds if it can output distinct inputs x, x' such that $H^s_{k_2}(x) = H^s_{k_2}(x')$, and we say that $(\widetilde{\mathsf{Gen}}, H)$ is *weakly collision resistant* if every PPT $\mathcal{A}$ succeeds in this experiment with only negligible probability. If $(\widetilde{\mathsf{Gen}}, H)$ is collision resistant then it is clearly weakly collision resistant as well; however, the latter is a weaker condition that is potentially easier to satisfy.

Weak collision resistance suffices for proving that NMAC is a secure MAC, as can be seen by careful examination of the proof sketch given previously. Note also that if s is a system-wide parameter (as is essentially the case when unkeyed hash functions are used), then a brute-force attack that finds a collision in H^s_{IV} for some value of IV does not help in finding a collision in $H^s_{k_2}$ for a randomly-chosen k_2. This is doubly true if k_2 is additionally kept secret — even if it is possible to find collisions in $H^s_{k_2}$ using, say, q invocations of the hash function (i.e., $q = 2^{n/2}$ if the birthday attack is used), it will be difficult for an adversary to obtain q legitimate tags from the honest communicating parties. To make an attack even more difficult, the adversary does not even receive $H^s_{k_2}(x)$; rather, it receives $h^s_{k_1}(H^s_{k_2}(x))$ that at the very least hides much of $H^s_{k_2}(x)$.

4.7.2 HMAC

A disadvantage of NMAC is that the IV of the underlying hash function H must be modified. In practice this may cause complications because the IV in, say, SHA-1 is fixed by the function specification and so existing cryptographic libraries do not enable an external IV input. HMAC solves this problem by keying the compression and hash functions differently. Another difference is that HMAC uses a single secret key, rather than two secret keys.

Let H and h be as in the previous section. We continue to assume that the output lengths of H and h are n bits, and that h compresses its input by half. We let IV be a *fixed* value for the initialization vector, that is assumed to be outside the control of the honest parties. We assume that when $x \in \{0,1\}^n$ (i.e., x is only a single block), then the computation of $H^s_{IV}(x)$ involves only a single invocation of the compression function h^s. This is technically not the case for our description of the Merkle-Damgård transform because we always appended the length as an additional block. However, in practice,

when the input is small enough, the length is included in the first block. In order to simplify these issues, when x is of length n, we will just ignore the concatenation of the length and set that $H^s(IV\|x) = h^s_{IV}(x)$.

CONSTRUCTION 4.17

Let $(\widetilde{\mathsf{Gen}}, h)$ be a fixed-length collision-resistant hash function, and let $(\widetilde{\mathsf{Gen}}, H)$ be the result of applying the Merkle-Damgård transform to $(\widetilde{\mathsf{Gen}}, h)$. Let IV, opad, and ipad be fixed constants of length n. HMAC defines a MAC as follows:

- Gen: on input 1^n, run $\widetilde{\mathsf{Gen}}(1^n)$ to obtain a key s. Also choose $k \leftarrow \{0,1\}^n$ at random. Output the key (s, k).
- Mac: on input a key (s, k) and a message $m \in \{0,1\}^*$ of length L, output the tag

$$t := H^s_{IV}\Big((k \oplus \mathsf{opad}) \,\|\, H^s_{IV}\big((k \oplus \mathsf{ipad}) \,\|\, m\big)\Big).$$

- Vrfy: on input a key (s, k), a message $m \in \{0,1\}^*$, and a tag t, output 1 if and only if $t \stackrel{?}{=} \mathsf{Mac}_{s,k}(m)$.

HMAC.

HMAC uses two constants opad and ipad. These are two strings of length n (i.e., the length of a single block of the input to H) and are defined as follows: opad consists of the byte "0x36" repeated as many times as needed; ipad is formed in the same way using the byte "0x5C".

HMAC is given as Construction 4.17 and is depicted graphically in Figure 4.4. At first glance, it looks very different from Construction 4.15. However, it is possible to view HMAC as a variant of NMAC. To see this, note that the first step in the computation of the inner hash $H^s_{IV}((k \oplus \mathsf{ipad})\|m)$ is to compute $k_2 \stackrel{\text{def}}{=} z_1 := h^s(IV \,\|\, (k \oplus \mathsf{ipad}))$. We therefore have that

$$H^s_{IV}((k \oplus \mathsf{ipad}) \,\|\, m) = H^s_{k_2}(m).$$

Likewise, the first step in the computation of the outer hash is to compute $k_1 \stackrel{\text{def}}{=} z'_1 := h^s(IV \,\|\, (k \oplus \mathsf{opad}))$. Thus:

$$H^s_{IV}\big((k \oplus \mathsf{opad}) \,\|\, H^s_{IV}\big((k \oplus \mathsf{ipad}) \,\|\, m\big)\big) = H^s_{k_1}(H^s_{k_2}(m)).$$

Since $H^s_{k_2}(m)$ is only a single block long, this in turn is equal to $h^s_{k_1}(H^s_{k_2}(m))$ (by our above assumption on H^s). Thus, this is exactly NMAC using the two *dependent* keys k_1, k_2. (These keys are dependent since they are derived in a deterministic way from the key k.)

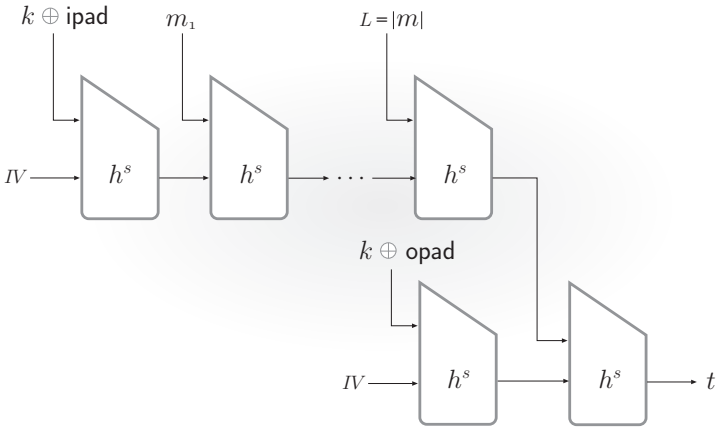

FIGURE 4.4: HMAC.

Define

$$G(k) \stackrel{\text{def}}{=} h^s\left(IV \| (k \oplus \mathsf{opad})\right) \| h^s\left(IV \| (k \oplus \mathsf{ipad})\right) = k_1 \| k_2. \tag{4.4}$$

If G is a pseudorandom generator and $k \leftarrow \{0,1\}^n$ is chosen uniformly at random, then the dependent keys k_1 and k_2 "look like" independent, uniformly-chosen keys and can be treated this way. Thus, if we assume that G is a pseudorandom generator then the security of HMAC reduces to the security of NMAC. We have the following theorem:

THEOREM 4.18 *Let* $(\widetilde{\mathsf{Gen}}, h)$ *satisfy the same conditions as in Theorem 4.16. If* G *as defined in Equation (4.4) is a pseudorandom generator, then HMAC is existentially unforgeable under an adaptive chosen-message attack (for arbitrary-length messages).*

HMAC in practice. HMAC is an industry standard and is widely used in practice. It is highly efficient and easy to implement, and is supported by a proof of security (based on assumptions that are believed to hold for practical hash functions that are considered collision resistant). The importance of HMAC is partially due to the timeliness of its appearance. Before the introduction of HMAC, many practitioners refused to use CBC-MAC (with the claim that it was "too slow") and instead used heuristic constructions of message authentication codes that were often insecure. For example, a common mistake was to define a MAC as $H^s(k\|x)$ (where k is a secret key). It is not difficult to show that when H is constructed using the Merkle-Damgård transform, this is not a secure MAC at all.

4.8 * Constructing CCA-Secure Encryption Schemes

In Section 3.7, we introduced the notion of CCA security for private-key encryption schemes. (We will not review the definition here, so the reader is encouraged to re-read that section before continuing.) In this section we will use message authentication codes, along with CPA-secure encryption schemes, to construct private-key encryption schemes meeting that notion of security.

Constructing CCA-secure encryption schemes. The construction works in the following way. The sender and receiver share two keys, one for a CPA-secure encryption scheme and the other for a message authentication code. To encrypt a message m, the sender first encrypts it using the CPA-secure scheme and then computes a MAC tag t on the resulting ciphertext c; the entire ciphertext is now $\langle c, t\rangle$. Given a ciphertext $\langle c, t\rangle$, the recipient verifies validity of the MAC tag before decrypting c.

Let us say a ciphertext $\langle c, t\rangle$ is *valid* if t is a valid MAC tag on c. The effect of this construction is that an adversary will be unable to generate *any* valid ciphertext that was not sent by one of the honest parties. (Technically speaking, we need an extra condition on the MAC; this, however, is the intuition.) This has the effect of rendering the decryption oracle useless, as we will see in the formal proof of security.

CONSTRUCTION 4.19

Let $\Pi_E = (\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme and let $\Pi_M = (\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ be a message authentication code. Define an encryption scheme $(\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ as follows:

- Gen': on input 1^n, run $\mathsf{Gen}_E(1^n)$ and $\mathsf{Gen}_M(1^n)$ to obtain keys k_1, k_2, respectively.
- Enc': on input a key (k_1, k_2) and a plaintext message m, compute $c \leftarrow \mathsf{Enc}_{k_1}(m)$ and $t \leftarrow \mathsf{Mac}_{k_2}(c)$ and output the ciphertext $\langle c, t\rangle$
- Dec': on input a key (k_1, k_2) and a ciphertext $\langle c, t\rangle$, first check whether $\mathsf{Vrfy}_{k_2}(c, t) \stackrel{?}{=} 1$. If yes, then output $\mathsf{Dec}_{k_1}(c)$; if no, then output $\perp$.

A CCA-secure private-key encryption scheme.

In Construction 4.19, we extend the syntax of a private-key encryption scheme so as to allow the decryption algorithm Dec' to output the special symbol $\perp$ indicating "failure". Correctness implies that no ciphertext output by Enc' results in the decryption algorithm outputting $\perp$.

Before proving security of this construction, we introduce an additional requirement on the MAC scheme. Say $(\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ has *unique tags* if for every k and every m there is a *unique* value t such that $\mathsf{Vrfy}_k(m, t) = 1$.

(This implies that Mac is deterministic, or might as well be.) The requirement of unique tags is not hard to achieve; in fact, the variable-length MAC of Construction 4.5 is the only construction we have seen that does *not* satisfy this property.

THEOREM 4.20 *If Π_E is a CPA-secure private-key encryption scheme and Π_M is a secure message authentication code with unique tags, then Construction 4.19 is a CCA-secure private-key encryption scheme.*

PROOF The idea behind the proof of this theorem is as follows. As defined earlier, say a ciphertext $\langle c, t\rangle$ is *valid* (with respect to secret key (k_1, k_2)) if $\mathsf{Vrfy}_{k_2}(c, t) = 1$. The adversary's queries to its decryption oracle are of two types: ciphertexts that the adversary received from its encryption oracle, and those that it did not. The first type of decryption query is not very useful, since the adversary already knows the message corresponding to the ciphertext. (Specifically, if the adversary received $\langle c, t\rangle$ in response to an encryption oracle query for the message m, then the adversary knows that the decryption of $\langle c, t\rangle$ is m.) As for ciphertexts of the second type, since Π_M is a secure message authentication code we can argue that (except with negligible probability) all such ciphertexts will be invalid and so the decryption oracle will simply return $\perp$ in this case. Thus, the second type of query is not useful either. Since the decryption oracle is essentially useless, security of $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ is reduced to the CPA-security of Π_E. We now give the formal proof.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary attacking Construction 4.19 in a chosen-ciphertext attack (cf. Definition 3.30). Let $\mathsf{ValidQuery}$ be the event that $\mathcal{A}$ submits a query $\langle c, t\rangle$ to its decryption oracle that was not previously obtained from its encryption oracle but for which $\mathsf{Vrfy}_{k_2}(c, t) = 1$. We have

$$\begin{aligned}&\Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1]\\ &\quad\leq \Pr[\mathsf{ValidQuery}] + \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}]. \qquad (4.5)\end{aligned}$$

We will show that $\Pr[\mathsf{ValidQuery}]$ is negligible, and that there exists a negligible function negl such that $\Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}] \leq \frac{1}{2} + \mathsf{negl}(n)$. This proves the theorem.

CLAIM 4.21 $\Pr[\mathsf{ValidQuery}]$ *is negligible.*

Intuitively, this is due to the fact that if $\mathsf{ValidQuery}$ occurs then the adversary has forged a valid tag t on a new "message" c. Formally, let $q(\cdot)$ be a polynomial that upper-bounds the number of decryption oracle queries made by $\mathcal{A}$, and consider the following adversary $\mathcal{A}_M$ attacking the message authentication code Π_M (i.e., running in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}_M,\Pi_M}(n)$):

Adversary $\mathcal{A}_M$:
$\mathcal{A}_M$ is given input 1^n and has access to an oracle $\mathsf{Mac}_{k_2}(\cdot)$.

1. Choose $k_1 \leftarrow \{0,1\}^n$ uniformly at random.
2. Choose $i \leftarrow \{1, \ldots, q(n)\}$ uniformly at random.
3. Run $\mathcal{A}$ on input 1^n. When $\mathcal{A}$ makes an encryption oracle query for the message m, answer it as follows:
   (i) Compute $c \leftarrow \mathsf{Enc}_{k_1}(m)$.
   (ii) Query c to the MAC oracle and receive t in response. Return $\langle c, t \rangle$ to $\mathcal{A}$.

   The challenge ciphertext is prepared in the exact same way (with a random bit $b \leftarrow \{0,1\}$ being chosen to select the message m_b that gets encrypted).
   When $\mathcal{A}$ makes a decryption oracle query for the ciphertext $\langle c, t \rangle$, answer it as follows:
   (i) If $\langle c, t \rangle$ was a response to a previous encryption oracle query for a message m, return m.
   (ii) If this is the ith decryption oracle query using a "new" value of c, output (c, t) and stop.
   (iii) Otherwise, return $\perp$.

In essence, $\mathcal{A}_M$ is "hoping" that the ith decryption oracle query of $\mathcal{A}$ will be the first query of the second type to be valid, in which case $\mathcal{A}_M$ outputs a valid forgery on a message c that it had never previously submitted to its MAC oracle.

Clearly $\mathcal{A}_M$ runs in probabilistic polynomial-time. We now analyze the probability that $\mathcal{A}_M$ generates a good forgery and so succeeds in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}_M,\Pi_M}(n)$. The key point is that the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}_M$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$, *until event* $\mathsf{ValidQuery}$ *occurs*. To see this, note that the encryption oracle queries of $\mathcal{A}$ (as well as computation of the challenge ciphertext) are simulated perfectly by $\mathcal{A}_M$. As for the decryption oracle queries of $\mathcal{A}$, until $\mathsf{ValidQuery}$ occurs these are all simulated properly: in case (i) this is obvious; in cases (ii) or (iii), if $\mathsf{ValidQuery}$ has not occurred then the correct answer to the decryption oracle query is indeed $\perp$. We conclude that the probability of event $\mathsf{ValidQuery}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}_M,\Pi_M}(n)$ is the same as the probability of this event in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$.

If $\mathcal{A}_M$ correctly guesses the index i representing the decryption query when $\mathsf{ValidQuery}$ first occurs, then $\mathcal{A}_M$ succeeds in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}_M,\Pi_M}(n)$. The probability that $\mathcal{A}_M$ correctly guesses i is $1/q(n)$. That is,

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}_M,\Pi_M}(n) = 1] \geq \Pr[\mathsf{ValidQuery}]/q(n).$$

Since Π_M is a secure MAC and q is polynomial, we conclude that $\Pr[\mathsf{ValidQuery}]$ is negligible. This completes the proof of Claim 4.21.

CLAIM 4.22 *There exists a negligible function* negl *such that*

$$\Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}] \leq \frac{1}{2} + \mathsf{negl}(n).$$

We now use the CPA-security of Π_E. Let $\mathcal{A}$ be as before, and consider the following adversary $\mathcal{A}_E$ attacking Π_E in a chosen-plaintext attack:

Adversary $\mathcal{A}_E$:
$\mathcal{A}_E$ is given input 1^n and has access to an oracle $\mathsf{Enc}_{k_1}(\cdot)$.

1. Choose $k_2 \leftarrow \{0,1\}^n$ uniformly at random.
2. Run $\mathcal{A}$ on input 1^n. When $\mathcal{A}$ makes an encryption oracle query for the message m, answer it as follows:
   (i) Query m to the encryption oracle and receive c in response.
   (ii) Compute $t \leftarrow \mathsf{Mac}_{k_2}(c)$, and return $\langle c, t\rangle$ to $\mathcal{A}$.

   When $\mathcal{A}$ makes a decryption oracle query for the ciphertext $\langle c, t\rangle$, answer it as follows:
   (i) If $\langle c, t\rangle$ was a response to a previous encryption oracle query for a message m, return m. Otherwise, return $\perp$.
3. When $\mathcal{A}$ outputs messages (m_0, m_1), output these same messages and receive a challenge ciphertext c in response. Compute $t \leftarrow \mathsf{Mac}_{k_2}(c)$, and return $\langle c, t\rangle$ as the challenge ciphertext for $\mathcal{A}$. Continue answering $\mathcal{A}$'s oracle queries as above.
4. Output the same bit b' that is output by $\mathcal{A}$.

Notice that $\mathcal{A}_E$ does not need a decryption oracle because it simply assumes that any decryption query by $\mathcal{A}$ that uses a "new" ciphertext $\langle c, t\rangle$ is invalid.

Clearly, $\mathcal{A}_E$ runs in probabilistic polynomial time. Furthermore, the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}_E$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$ *as long as event* $\mathsf{ValidQuery}$ *never occurs.* (This is because if $\mathsf{ValidQuery}$ never occurs, then the correct response to any decryption query by $\mathcal{A}$ that uses a new value of c is indeed $\perp$.) Therefore, the probability that $\mathcal{A}_E$ succeeds when $\mathsf{ValidQuery}$ does not occur is the same as the probability that $\mathcal{A}$ succeeds when $\mathsf{ValidQuery}$ does not occur; i.e.,

$$\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}] = \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}],$$

implying that

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1] &\geq \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}] \\ &= \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}].\end{aligned}$$

Since Π_E is CPA-secure, there exists a negligible function negl such that $\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$, implying Claim 4.22. The theorem is derived by combining Claims 4.21 and 4.22 with Equation (4.5). ∎

The role of unique tags. If the MAC scheme Π_M does not have unique tags, then Construction 4.19 may not be CCA-secure. Specifically, it may be easy to modify a valid tag t on a value c into a different (but still valid) tag t' on the same value. If the challenge ciphertext is $\langle c, t \rangle$, then an adversary can query $\langle c, t' \rangle$ to its decryption oracle and thus potentially learn the plaintext message m_b.

A weaker condition than the unique tags property suffices for the proof of the previous theorem: it suffices that an adversary cannot *find* a different valid tag t' on a previously-authenticated message m (though different tags may exist). Secure message authentication codes with this property are said to be *strongly-secure*. In any case, there exist numerous efficient MACs that have unique tags, so requiring it is not much of a restriction.

CCA-security and real-life implications. Construction 4.19 may seem unsatisfying, since it achieves CCA-security simply by ensuring that the decryption oracle is useless to the adversary. Instead of being viewed as a drawback of the construction, this should be viewed as an advantage! Specifically, although there are other ways to achieve CCA-security (see Exercise 3.14 for an example), here the adversary is unable to generate *any* valid ciphertext that was not already created by one of the honest parties. As we will see in the next section, this means that Construction 4.19 achieves both privacy *and* message authentication.

4.9 * Obtaining Privacy and Message Authentication

In Chapter 3, we studied how it is possible to encrypt messages and thereby obtain *privacy*. In this chapter, we have shown how message authentication codes can be used to guarantee *data authenticity* or *integrity*. Often, however, we need both privacy and message integrity. It may be tempting to think that any combination of a secure encryption scheme and a secure message authentication code should provide both of these properties. Unfortunately, this is not at all the case. In general, even excellent cryptographic tools can be combined in such a way that the result is insecure. Thus, unless a specific combination has been proven secure, one should exercise care in using it.

There are three common approaches to combining encryption and message authentication. Let k_1 denote an encryption key, and let k_2 be a MAC key.[9]

[9] A common mistake is to use the *same* key for both encryption and authentication. This should never be done, as independent keys should always be used for independent applications (unless a specific proof of security when using the same key is known). Further discussion of this point is given at the end of this section.

The three approaches are:

1. *Encrypt-and-authenticate:* In this method, encryption and message authentication are computed independently. That is, given a plaintext message m, the sender transmits $\langle c, t\rangle$ where:

$$c \leftarrow \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t \leftarrow \mathsf{Mac}_{k_2}(m).$$

   The receiver decrypts c to recover m, and then verifies the tag t. If $\mathsf{Vrfy}_{k_2}(m, t) = 1$, the receiver outputs m; otherwise, it outputs $\perp$.

2. *Authenticate-then-encrypt:* Here a MAC tag t is first computed, and then the message and tag are encrypted together. That is, the sender transmits c computed as:

$$t \leftarrow \mathsf{Mac}_{k_2}(m) \quad \text{and} \quad c \leftarrow \mathsf{Enc}_{k_1}(m \| t).$$

   The authentication tag t is not sent "in the clear", but is instead incorporated into the plaintext that is encrypted. The receiver decrypts c, and then verifies the tag t on m. As before, if $\mathsf{Vrfy}_{k_2}(m, t) = 1$ the receiver outputs m; otherwise, it outputs $\perp$.

3. *Encrypt-then-authenticate:* In this case, the message m is first encrypted and then a MAC tag is computed over the encrypted message. That is, the message is the pair $\langle c, t\rangle$ where:

$$c \leftarrow \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t \leftarrow \mathsf{Mac}_{k_2}(c).$$

   The receiver verifies t before decrypting c. Observe that this is exactly Construction 4.19 from the previous section.

In this section we analyze each of the above approaches when instantiated using an *arbitrary* CPA-secure encryption scheme and an *arbitrary* secure MAC (with unique tags). Our analysis will follow an *all or nothing* approach: that is, we will only be satisfied with a combination that provides both privacy and integrity when using *any* CPA-secure encryption scheme and *any* secure message authentication code. In other words, we will reject any combination for which there exists even a single counterexample of a secure encryption scheme/MAC for which the combination is insecure. As an example, we will reject the "encrypt-and-authenticate" approach as being insecure. This does *not* mean that for every secure encryption scheme and MAC the "encrypt-and-authenticate" approach results in an insecure scheme. Rather it means that there *exists* a secure encryption scheme and a MAC for which the combination is insecure.

The reason we insist on an "all or nothing" approach is that this reduces the likelihood of errors in implementation. This is due to the fact that it should be possible to replace any secure encryption scheme with another one (and

likewise for the MAC) without affecting the security of applications that use the scheme. Such replacements are common in practice when cryptographic libraries are updated, or when standards are modified.

Privacy only vs. privacy and message integrity. Most online tasks, and clearly any online purchase or bank transaction, needs to be both encrypted and authenticated. In general, however, it is not always clear when authentication is needed in addition to secrecy. For example, when encrypting files on a disk, is it necessary to also authenticate them? At first sight, one may think that since disk encryption is used to prevent an attacker from reading secret files, there is no need for authentication. However, it may be possible for an adversary to inflict significant damage if financial reports and so on are modified (e.g., thereby causing a company to mistakenly publish false reports). It is best practice to *always encrypt and authenticate by default*; encryption alone should not be used unless there are compelling reasons to do so (such as implementations on severely resource-constrained devices) and, even then, only if one is absolutely sure that no damage can be caused by undetected modification of the data. Note that lack of integrity can sometimes lead to a breach of privacy, as we have seen for the case of chosen-ciphertext attacks.

Security requirements. In order to analyze which of the combinations of encryption and authentication are secure, we must first define what we mean by a "secure combination". The best approach for this is to model in general what we mean by a *secure communication channel* and then prove that a given combination meets this definition. Unfortunately, providing a formal definition of a secure channel is beyond the scope of this book. We therefore provide a more "naive" definition that simply incorporates notions of privacy and message integrity separately. This definition and the resulting analysis suffice for understanding the key issues at hand.

Let $\Pi_E = (\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ be an arbitrary encryption scheme and let $\Pi_M = (\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ be a message authentication code. A *message transmission scheme* $\Pi' = (\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}')$ derived as a combination of Π_E and Π_M is a tuple of algorithms that operate as follows:

- The *key-generation algorithm* Gen' takes input 1^n, and runs $\mathsf{Gen}_E(1^n)$ and $\mathsf{Gen}_M(1^n)$ to obtain keys k_1 and k_2, respectively. The key is (k_1, k_2).

- The *message transmission algorithm* EncMac' takes as input the keys (k_1, k_2) and a message m and outputs a value c that is derived by applying some combination of $\mathsf{Enc}_{k_1}(\cdot)$ and $\mathsf{Mac}_{k_2}(\cdot)$.

- The *decryption algorithm* Dec' takes as input the keys (k_1, k_2) and a transmitted value c, and applies some combination of $\mathsf{Dec}_{k_1}(\cdot)$ and $\mathsf{Vrfy}_{k_2}(\cdot)$. The output of Dec' is either a plaintext m or a special symbol $\perp$ that indicates an error.

The correctness requirement is that for every n, every pair of keys (k_1, k_2) output by $\mathsf{Gen}'(1^n)$, and every value $m \in \{0,1\}^*$,

$$\mathsf{Dec}'_{k_1,k_2}(\mathsf{EncMac}'_{k_1,k_2}(m)) = m.$$

Π' actually satisfies the syntax of a private-key encryption scheme. We refer to it as a "message transmission scheme" only because when we define security we will require message integrity in addition to privacy.

As we have mentioned, we will define security for a message transmission scheme Π' by defining separate notions of privacy and authenticity. The notion of privacy we consider is one we have seen before: that Π' (now viewed as an encryption scheme) be CCA-secure. (Looking ahead, this will have the effect of "boosting" the privacy requirement, because we will start with an encryption scheme Π_E that is only CPA-secure.) Our notion of message integrity will be essentially that of existential unforgeability under an adaptive chosen-message attack. Since Π' does not satisfy the syntactic requirements of a message authentication code, however, we introduce a definition specific to this case. Consider the following experiment defined for a message transmission scheme Π', adversary $\mathcal{A}$, and value n for the security parameter:

The secure message transmission experiment $\mathsf{Auth}_{\mathcal{A},\Pi'}(n)$**:**

1. *A random key* $k = (k_1, k_2)$ *is generated by running* $\mathsf{Gen}'(1^n)$.
2. *The adversary* $\mathcal{A}$ *is given input* 1^n *and oracle access to the message transmission algorithm* $\mathsf{EncMac}'_k(\cdot)$. *The adversary eventually outputs* c. *Let* $\mathcal{Q}$ *denote the set of all queries that* $\mathcal{A}$ *asked to its oracle.*
3. *Let* $m := \mathsf{Dec}'_k(c)$. *The output of the experiment is defined to be 1 if and only if* **(1)** $m \neq \perp$ *and* **(2)** $m \notin \mathcal{Q}$.

DEFINITION 4.23 *A message transmission scheme* Π' *achieves* authenticated communication *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$, *there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{Auth}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

Comparing the above experiment to the message authentication experiment immediately preceding Definition 4.2, we see that the adversary's job is somewhat easier here, since the adversary does not need to know the message m to which its output c corresponds. This means that constructions satisfying the above definition are (in some sense) more secure than constructions satisfying only Definition 4.2.

With the above in place, we can now define a secure message transmission scheme.

DEFINITION 4.24 *A message transmission scheme* $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}')$ *is* secure *if it is both a CCA-secure encryption scheme and also achieves authenticated communication.*

We now analyze the three approaches discussed previously for combining encryption and authentication.

Encrypt-and-authenticate. As we have described, in this approach encryption and message authentication are computed separately. Given a message m, the transmitted value is $\langle c, t \rangle$ where

$$c \leftarrow \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t \leftarrow \mathsf{Mac}_{k_2}(m).$$

This combination is *not* (necessarily) secure, since it may violate privacy. To see this, note that a secure MAC does not necessarily imply *any* privacy and, specifically, it is possible for the tag of a message to leak the entire message. (In particular, if $(\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ is a secure message authentication code, then so is the scheme defined by $\mathsf{Mac}'_k(m) = (m, \mathsf{Mac}_k(m))$.) So the encrypt-and-authenticate combination may yield a scheme that does not even have indistinguishable encryptions in the presence of an eavesdropper, the most basic level of privacy.

The above may seem to be an unnatural counterexample, but is enough because our requirements were that security should hold for *any* secure instantiation of the underlying building blocks. In any case, it is not hard to see that the encrypt-and-authenticate combination is not secure even when natural MACs are used (for any choice of underlying encryption scheme). In Exercise 4.19 you are asked to show that, e.g., if CBC-MAC is used as Π_M then security against chosen-plaintext attacks does not hold.

Authenticate-then-encrypt. Here, a MAC tag $t \leftarrow \mathsf{Mac}_{k_2}(m)$ is first computed; then $m \| t$ is encrypted, and the resulting value $\mathsf{Enc}_{k_1}(m \| \mathsf{Mac}_{k_2}(m))$ is transmitted. We show that this combination is also not necessarily secure. We use the following encryption scheme:

- Let $\mathsf{Transform}(m)$ be as follows: any 0 in m is transformed to 00, and any 1 in m is transformed arbitrarily to 01 or 10.[10] The inverse of this transform parses the encoded message as pairs of bits, and then maps 00 to 0, and 01 or 10 to 1. If a 11 is encountered, the result is $\perp$. (That is, $\mathsf{Transform}^{-1}(01\,10) = 11$ but $\mathsf{Transform}^{-1}(01\,11) = \perp$.)
- Define $\mathsf{Enc}_k(m) = \mathsf{Enc}'_k(\mathsf{Transform}(m))$, where Enc' represents counter mode encryption using a pseudorandom function. (The important point is that Enc' works by generating a new pseudorandom stream for each

[10] Of course, this encoding is contrived. However, encodings of inputs are often used and it would certainly be undesirable if the security of a cryptographic scheme depended on which encoding were used.

message to encrypt, and then XORing the stream with the message.) Note that Enc is CPA-secure.

We show that the authenticate-then-encrypt combination of the above encryption scheme with *any* MAC is not secure against a chosen-ciphertext attack. The attack we show works as long as an adversary can find out if a given ciphertext is valid, even if it cannot obtain the entire decryption of the ciphertext. Thus, the attack may be easy to carry out, as all an adversary needs to do is forward a ciphertext to one of the parties and observe whether their reaction is consistent with an invalid ciphertext (e.g., if the party requests re-transmission) or not.

Consider the following chosen-ciphertext attack: Given a challenge ciphertext $c = \mathsf{Enc}'_{k_1}(\mathsf{Transform}(m \| \mathsf{Mac}_{k_2}(m)))$, the attacker simply flips the first two bits of the second block of c (recall that the first block of c is an initial counter value ctr) and verifies whether the resulting ciphertext is valid. (This can be determined via a query to the decryption oracle. The adversary only needs to know if the ciphertext is valid, however, and so a weaker oracle would also suffice.) If the first bit of the underlying message m is 1, then the modified ciphertext will be valid. This is because if the first bit of m is 1, then the first two bits of $\mathsf{Transform}(m)$ are 01 or 10, and flipping these bits yields another valid encoding of m. (Our choice of Enc' ensures that flipping the first two bits of the ciphertext has the effect of flipping the first two bits of the encoded message.) Furthermore, the tag will still be valid since it is applied to m and not the encoding of m. On the other hand, if the first bit of m is 0 then the modified ciphertext will not be valid since the first two bits of $\mathsf{Transform}(m)$ would be 00 and their complement would be 11.

This attack can be carried out on each bit of m separately, resulting in complete recovery of the message m.

This counterexample demonstrates that the authenticate-then-encrypt combination is not, in general, secure. However, some specific instantiations of this approach are secure; an example is the specific combination used within SSL. Nevertheless, as mentioned previously, it is bad practice to use a methodology whose security depends on specific implementations.

Encrypt-then-authenticate. In this approach, the message is first encrypted and then a MAC is computed over the ciphertext. That is, the message is the pair $\langle c, t \rangle$ where

$$c \leftarrow \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t \leftarrow \mathsf{Mac}_{k_2}(c).$$

Decryption is done as in Construction 4.19. We have the following theorem:

THEOREM 4.25 *Let Π_E be a CPA-secure private-key encryption scheme, and let Π_M be a secure message authentication code with unique tags. Then the combination $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}')$ derived by applying the encrypt-then-authenticate approach to Π_E, Π_M is a secure message transmission scheme.*

We have already proved that the above combination is CCA-secure in Theorem 4.20, and leave as an exercise the proof that it also provides authenticated communication.

Secure message transmission vs. CCA-security. Although we use the same construction for achieving CCA-security and secure message transmission, the security goals in each case are different. In the setting of CCA-security we are not necessarily interested in obtaining message authentication; rather, we wish to ensure privacy even against a strong adversary who is able to make decryption queries. When considering secure message transmission, in contrast, we are interested in the twin goals of CCA-security and integrity. Clearly, as we have defined it, secure message transmission implies CCA-security. The opposite direction is not necessarily true.

The need for independent keys. We conclude by stressing a basic principle of security and cryptography: *different security goals should always use different keys.* That is, if an encryption scheme and a message authentication code are both needed, then independent keys should be used for each one. In order to illustrate this here, consider what can happen to the encrypt-then-authenticate methodology when the same key k is used for both encryption and authentication. Let F be a strong pseudorandom permutation. It follows that both F and F^{-1} are strong pseudorandom permutations. Define $\mathsf{Enc}_k(m) = F_k(m\|r)$ for $m \in \{0,1\}^{n/2}$ and a random $r \leftarrow \{0,1\}^{n/2}$, and define $\mathsf{Mac}_k(c) = F_k^{-1}(c)$. It can be shown that the given encryption scheme is CPA-secure (in fact, it is even CCA-secure), and we know that the given message authentication code is a secure MAC. However, the encrypt-then-authenticate combination applied to the message m with the same key k yields:

$$\mathsf{Enc}_k(m), \mathsf{Mac}_k(\mathsf{Enc}_k(m)) = F_k(m\|r), F_k^{-1}(F_k(m\|r)) = F_k(m\|r), m\|r,$$

and the message m is revealed in the clear!

References and Additional Reading

The definition of security for message authentication codes was adapted by Bellare et al. [12] from the definition of security for digital signatures given by Goldwasser et al. [70] (see Chapter 12). The basic paradigm of using pseudorandom functions for message authentication (as in Construction 4.3) was introduced by Goldreich et al. [66], and Construction 4.5 for extending a fixed-length MAC to a variable-length MAC is due to Goldreich [65]. An alternate approach for extending a fixed-length MAC to a variable-length MAC using collision-resistant hash functions is presented in the context of digital signatures in Section 12.4 (and, as we have mentioned, is the same idea underlying NMAC).

CBC-MAC was standardized in the early '80s [148, 80] and was later formally analyzed and proven secure by Bellare et al. [12] (the proofs include both the fixed-length case and the secure extensions to variable-length messages). The NMAC and HMAC constructions were introduced by Bellare et al. [8] and later standardized in [110].

Collision-resistant hash functions were first formally defined by Damgård [43], and the Merkle-Damgård transform was introduced independently by Damgård and Merkle [44, 102]. Additional discussion regarding notions of security for hash functions besides collision resistance can be found in [99, 123]. For further information about SHA-1 and MD5, see, e.g., the textbook by Kaufman et al. [87]. Note, however, that their treatment pre-dates the recent attacks by Wang et al. [144, 143]. For other interesting applications of collision-resistant hash functions in computer security, see Kaufman et al. [87] (but beware that in many cases the security arguments they give are purely heuristic). There are many hash functions that appear in the literature; many have been broken and some have not. Up-to-date information is maintained at the "Hash Function Lounge" [7].

The method of encrypting and then applying a MAC in order to achieve CCA-security was described by Dolev et al. [50]. Bellare and Namprempre [13] and Krawczyk [90] analyze different methods for simultaneously achieving privacy and authentication, and Krawczyk also analyzes the authenticate-then-encrypt approach used within SSL. Other notions of secure encryption that incorporate integrity are discussed in [85, 13]. A solution to Exercise 4.5 is given in [11].

Exercises

4.1 Say $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ is a secure MAC, and for $k \in \{0,1\}^n$ the tag-generation algorithm Mac_k always outputs tags of length $t(n)$. Prove that t must be super-logarithmic or, equivalently, that if $t(n) = \mathcal{O}(\log n)$ then Π cannot be a secure MAC.

Hint: Consider the probability of randomly guessing a valid tag.

4.2 Consider the following fixed-length MAC for messages of length $\ell(n) = 2n - 2$ using a pseudorandom function F: On input a message $m_0 \| m_1$ (with $|m_0| = |m_1| = n - 1$) and key $k \in \{0,1\}^n$, algorithm Mac outputs $t = F_k(0\|m_0)\|F_k(1\|m_1)$. Algorithm Vrfy is defined in the natural way. Is $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ existentially unforgeable under a chosen-message attack? Prove your answer.

4.3 Let F be a pseudorandom function. Show that the following MAC for messages of length $2n$ is insecure: The shared key is a random $k \in \{0,1\}^n$. To authenticate a message $m_1\|m_2$ with $|m_1| = |m_2| = n$, compute the tag $\langle F_k(m_1), F_k(F_k(m_2))\rangle$.

4.4 Let F be a pseudorandom function. Show that each of the following message authentication codes is insecure. (In each case the shared key is a random $k \in \{0,1\}^n$.)

(a) To authenticate a message $m = m_1 \| \cdots \| m_\ell$, where $m_i \in \{0,1\}^n$, compute $t := F_k(m_1) \oplus \cdots \oplus F_k(m_\ell)$.

(b) To authenticate a message $m = m_1 \| \cdots \| m_\ell$, where $m_i \in \{0,1\}^n$, choose $r \leftarrow \{0,1\}^n$ at random, compute $t := F_k(r) \oplus F_k(m_1) \oplus \cdots \oplus F_k(m_\ell)$, and send $\langle r, t \rangle$.

(c) To authenticate a message $m = m_1 \| \cdots \| m_\ell$, where $m_i \in \{0,1\}^{n/2}$, choose $r \leftarrow \{0,1\}^n$ at random, compute

$$t := F_k(r) \oplus F_k(\langle 1 \rangle \| m_1) \oplus \cdots \oplus F_k(\langle \ell \rangle \| m_\ell)$$

(where $\langle i \rangle$ is an $n/2$-bit encoding of the integer i), and send $\langle r, t \rangle$.

4.5 Consider an extension of the definition of secure message authentication where the adversary is provided with both a Mac and a Vrfy oracle.

(a) Provide a formal definition of security in this case, and explain what real-world adversarial actions are modeled by providing the adversary with a Vrfy oracle.

(b) Show that if Π has unique tags (cf. Section 4.8), then Π satisfies your definition if it satisfies Definition 4.2.

(c) Show that if Π does *not* have unique tags, then Π may satisfy Definition 4.2 but not your definition.

4.6 Is Construction 4.3 necessarily secure when instantiated using a weak pseudorandom function (cf. Exercise 3.20)? Explain.

4.7 Prove that Construction 4.5 is secure if it is changed as follows: Instead of including ℓ in every block, set $t_i := F_k(r \| b \| i \| m_i)$ where b is a single bit such that $b = 0$ in all blocks but the last one, and $b = 1$ in the last block. What is the advantage of this modification?

4.8 Show that the basic CBC-MAC construction is *not* secure when used to authenticate messages of different lengths.

4.9 Prove that the following modifications of CBC-MAC do not yield a secure fixed-length MAC:

(a) Modify CBC-MAC so that a random IV is used each time a tag is computed (and the IV is output along with t_ℓ). I.e., $t_0 \leftarrow \{0,1\}^n$ is chosen uniformly at random rather than being fixed to 0^n, and the tag is t_0, t_ℓ.

(b) Modify CBC-MAC so that all blocks $t_1, \ldots, t_\ell$ are output (rather than just t_ℓ).

4.10 Provide formal definitions for second pre-image resistance and pre-image resistance. Formally prove that any hash function that is collision resistant is second pre-image resistant, and that any hash function that is second pre-image resistant is pre-image resistant.

4.11 Let (Gen_1, H_1) and (Gen_2, H_2) be two hash functions. Define (Gen, H) so that Gen runs Gen_1 and Gen_2 to obtain keys s_1 and s_2, respectively. Then define $H^{s_1,s_2}(x) = H^{s_1}(x) \| H^{s_2}(x)$.

(a) Prove that if at least one of (Gen_1, H_1) and (Gen_2, H_2) is collision resistant, then (Gen, H) is collision resistant.

(b) Determine whether an analogous claim holds for second pre-image resistance and pre-image resistance, respectively. Prove your answer in each case.

4.12 Let (Gen, H) be a collision-resistant hash function. Is $(\mathsf{Gen}, \hat{H})$ defined by $\hat{H}^s(x) \stackrel{\text{def}}{=} H^s(H^s(x))$ necessarily collision resistant?

4.13 Provide a formal proof of Theorem 4.14 (i.e., describe the formal reduction).

4.14 Generalize the Merkle-Damgård construction for any compression function that compresses by at least one bit. You should refer to a general input length ℓ' and general output length ℓ (with $\ell' > \ell$).

4.15 For each of the following modifications to the Merkle-Damgård transform, determine whether the result is collision resistant or not. If yes, provide a proof; if not, demonstrate an attack.

(a) Modify the construction so that the input length is not included at all (i.e., output z_B and not $z_{B+1} = h^s(z_B \| L)$).

(b) Modify the construction so that instead of outputting $z = h^s(z_B \| L)$, the algorithm outputs $z_B \| L$.

(c) Instead of using a fixed IV, choose $IV \leftarrow \{0,1\}^n$ and define $z_0 := IV$. Then, set the output to be $IV \,\|\, h^s(z_B \| L)$.

(d) Instead of using an IV, just start the computation from x_1. That is, define $z_1 := x_1$ and then compute $z_i := h^s(z_{i-1} \| x_i)$ for $i = 2, \ldots, B+1$ and output z_{B+1} as before.

(e) Instead of using a fixed IV, set $z_0 := L$ and then compute $z_i := h^s(z_{i-1} \| x_i)$ for $i = 1, \ldots, B$ and output z_B.

4.16 Provide a full and detailed specification of HMAC when the underlying compression function has input length ℓ' and output length ℓ (with $\ell' > \ell$). Describe the instantiation of HMAC with SHA-1.

4.17 Before HMAC was invented, it was quite common to define a MAC by $\mathsf{Mac}_k(m) = H^s(k\|m)$ where H is a collision-resistant hash function. Show that this is not a secure MAC when H is constructed via the Merkle-Damgård transform.

4.18 Show that Construction 4.19 is CCA-secure even when the MAC of Construction 4.5 is used (this MAC does not have unique tags).

4.19 Show that if any message authentication code having unique tags is used in the encrypt-and-authenticate approach, the resulting combination is not CPA-secure.

4.20 Show an encryption scheme that is CCA-secure but is not a secure message transmission scheme.

4.21 Show a message transmission scheme that achieves authenticated communication but is not a secure message transmission scheme.

4.22 Prove Theorem 4.25.

Chapter 5

Practical Constructions of Pseudorandom Permutations (Block Ciphers)

In previous chapters, we have studied how pseudorandom permutations can be used to construct secure encryption schemes and message authentication codes. However, one question of prime importance that we have not yet studied is how pseudorandom permutations are constructed in the first place, or even whether they exist at all! In the next chapter we will study these questions from a theoretical vantage point, and show constructions of pseudorandom permutations that can be proven secure based on quite weak assumptions. In this chapter, our focus will be on comparatively heuristic, but far more efficient, constructions of pseudorandom permutations — known as *block ciphers* — that are used in practice.

As just mentioned, the constructions of block ciphers that we will explore in this chapter are (for the most part) heuristic, at least in the sense that they have no known proof of security based on any weaker assumption. Nevertheless, a number of the block ciphers that are used in practice have withstood many years of public scrutiny and attempted cryptanalysis, and given this fact it is quite reasonable to assume that these block ciphers are indeed (strong) pseudorandom permutations, subject to the technical issues discussed below.

Of course, in some sense there is no fundamental difference between assuming, say, that factoring is hard and assuming that DES (a block cipher we will study in detail later in this chapter) is a pseudorandom permutation. There is, however, a significant *qualitative* difference between these assumptions.[1] The primary difference is that the former assumption is of a weaker type: That is, the requirement that a certain problem (i.e., factoring) be hard to solve seems "easier to satisfy" than the requirement that a given keyed function be indistinguishable from a random function. Less important but still relevant differences between the assumptions are that the problem of factoring has been studied much longer than the problem of distinguishing DES from a random function, and the fact that factoring was recognized as a hard mathematical problem well before the advent of cryptographic schemes based

[1] It should be clear that the discussion in this paragraph is informal, as we cannot formally argue about any of this when we cannot even prove that factoring is hard in the first place!

on it. We remark further that most of the cryptanalytic effort directed at DES and other block ciphers has focused on *key-recovery attacks*, where the goal is to recover the key k given multiple pairs $(x, DES_k(x))$; comparatively less research has been specifically aimed at the (potentially easier) problem of distinguishing DES from a random function. (This is perhaps less true with more modern block ciphers; see below.)

To summarize, the assumption that a well-studied block cipher such as DES is pseudorandom is indeed a reasonable one, and one that people are comfortable relying on in practice. Still, it would be preferable to be able to base security of modern-day block ciphers on weaker and more long-standing assumptions. As we will see in Chapter 6, this is (in principle) possible; unfortunately, the constructions we will see there are orders of magnitude less efficient than the block ciphers in use today.

Block Ciphers as Strong Pseudorandom Permutations

A block cipher is an efficient, keyed permutation $F : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^\ell$. Recall from Section 3.6.3 that this means the function F_k defined by $F_k(x) = F(k,x)$ is a bijection (i.e., a permutation), and moreover F_k and its inverse F_k^{-1} are efficiently computable given k. We refer to n as the *key length* and ℓ as the *block length* of F. Departing from our convention in Chapter 3, we no longer require that $n = \ell$ (though that will often be the case). A more important difference is that here n and ℓ are fixed constants, whereas in Chapter 3 they were viewed as functions of a security parameter n. This essentially puts us in a setting of concrete security rather than asymptotic security, as discussed further below.[2]

Despite their name, block ciphers should be viewed as (strong) pseudorandom permutations and *not* as encryption schemes. Stated differently, block ciphers should be viewed as *building blocks* for encryption and other schemes, and not as encryption schemes themselves. As discussed in Chapter 3, modeling block ciphers as (strong) pseudorandom permutations allows rigorous proofs of security for constructions based on block ciphers, and also makes explicit the necessary requirements on the block cipher. Moreover, a solid understanding of what block ciphers are supposed to achieve (e.g., as per Definition 3.28) is instrumental in their design. The view that block ciphers should be modeled as pseudorandom permutations has, at least in the recent past, served as a major influence in their design. As an example, the call for proposals for the recent Advanced Encryption Standard (AES) that we will encounter later in this chapter stated the following evaluation criteria:

[2] Though a block cipher with fixed key length has no "security parameter" to speak of, we still view security as depending on the length of the key and thus denote this value by n. We remark that viewing the key length as a parameter makes sense when comparing block ciphers having different key lengths, or when using a block cipher that supports keys of different lengths.

> *The security provided by an algorithm is the most important factor.... Algorithms will be judged on the following factors...*
>
> - *The extent to which the algorithm output is indistinguishable from a random permutation on the input block.*

Essentially, this states that a block cipher should be a *pseudorandom permutation*. (It is unclear to what extent submitted proposals were evaluated as *strong* pseudorandom permutations. Nevertheless, had an attack been demonstrated showing that some proposal did not satisfy this criterion, it is unlikely the proposal would have been adopted.) Thus, as we have stated, modern block ciphers are intended to be pseudorandom permutations. As such, they are suited for use in all the constructions relying on pseudorandom permutations (or pseudorandom functions) that we have seen in this book.

Block ciphers and Definition 3.28. Although we treat block ciphers as pseudorandom permutations, a technical difficulty is that any fixed block cipher is typically only defined for a fixed key length and block length. This means that an asymptotic definition of security as in Definition 3.28 does not apply. A similar problem arose (though we did not dwell on it there) when we considered practical constructions of collision-resistant hash functions such as SHA-1. In both cases, the appropriate way to deal with this is to understand that block ciphers are only intended to be indistinguishable from random for attackers running "for any reasonable amount of time" (say, 100 years using the strongest available computers). This can be treated formally within the framework of *concrete security*, briefly mentioned in Section 3.1.1.

Attacks on Block Ciphers

Notwithstanding the fact that block ciphers should not be confused with encryption schemes, the standard terminology (that we will adopt here) for attacks on a block cipher F refers to:

- *Ciphertext-only attacks*, where the attacker is given only a series of outputs $\{F_k(x_i)\}$ for some inputs $\{x_i\}$ unknown to the attacker
- *Known-plaintext attacks*, where the attacker is given pairs of inputs and outputs $\{(x_i, F_k(x_i))\}$
- *Chosen-plaintext attacks*, where the attacker is given $\{(x_i, F_k(x_i)\}$ for a series of inputs $\{x_i\}$ that are chosen by the attacker
- *Chosen-ciphertext attacks*, where the attacker is given $\{(x_i, F_k(x_i)\}$ and $\{(F_k^{-1}(y_i), y_i)\}$ for $\{x_i\}, \{y_i\}$ chosen by the attacker.

In view of the above, a pseudorandom permutation is secure against chosen-plaintext attacks, while a strong pseudorandom permutation is secure against

chosen-ciphertext attacks. In an asymptotic sense, this refers to attacks running in polynomial time. As mentioned just previously, though, block ciphers typically have some fixed block length and key length, and so asymptotic measures are useless. Instead, the goal is for a block cipher to be unbreakable in any reasonable amount of time. This is interpreted very strictly in the context of block ciphers, and a block cipher is generally only considered"good" if the best known attack has time complexity roughly equivalent to a brute-force search for the key. Thus, if a cipher with key length $n = 112$ can be broken in time 2^{56} (we will see such an example later), the cipher is (generally) considered insecure even though 2^{56} is still a relatively large number. Note that in an asymptotic setting, an attack of complexity $2^{n/2}$ is not feasible since it requires exponential time (and thus a cipher where such an attack is possible might still satisfy the definition of being a pseudorandom permutation). In a non-asymptotic setting, however, we must worry about the actual time complexity of the attack (rather than its asymptotic behavior). Furthermore, we are concerned that existence of such an attack may indicate some more fundamental weakness in the design of the cipher.

The Aim of this Chapter

To head off any confusion, we stress that the main aim of this chapter is to present some design principles used in the construction of modern block ciphers, with a secondary aim being to introduce the reader to the popular block ciphers DES and AES. We caution the reader that:

- It is *not* our intent to present the low-level details of DES or AES, and our description of these block ciphers should not be relied upon for implementation. To be clear: our descriptions of these ciphers are often (purposefully) inaccurate, as we omit certain technical details when they are not relevant to the broader point we are trying to emphasize.

- It is also *not* the aim of this chapter to teach how to construct secure block ciphers. On the contrary, we strongly believe that new (and proprietary) block ciphers should neither be constructed nor used since numerous excellent block ciphers are readily available.

  Those who are interested in developing expertise in constructing block ciphers are advised to start with the references at the end of this chapter.

5.1 Substitution-Permutation Networks

The main property required of a block cipher is that it should behave like a random permutation. Of course, a truly random permutation would be

perfect. However, a random permutation having a *block length* (i.e., input and output length) of n bits would require $\log(2^n!) \approx n \cdot 2^n$ bits for its representation, something that is impractical for $n > 20$ and completely infeasible for $n > 50$. (Looking ahead, in practice a block length of $n \approx 64$ is already too small in some cases, and modern block ciphers thus have block lengths of $n \geq 128$.) Thus, we need to somehow construct a *concise* function that behaves like a random one.

The confusion-diffusion paradigm. In addition to his work on perfect secrecy, Shannon introduced a basic paradigm for constructing concise random-looking permutations. The basic idea is to construct a random-looking permutation F with a large block length from many smaller random (or random-looking) permutations $\{f_i\}$ having a small block length. Let us see how this works on the most basic level. Say we want F to have a block length of 128 bits. We can define F as follows: the key k for F will specify 16 *random* permutations $f_1, \ldots, f_{16}$ that each have an 8-bit block length.[3] Given an input $x \in \{0,1\}^{128}$, we parse it as 16 consecutive 8-bit blocks $x_1 \cdots x_{16}$ and then set

$$F_k(x) = f_1(x_1) \cdots f_{16}(x_{16}). \tag{5.1}$$

We say (informally) that these $\{f_i\}$ introduce *confusion* into F.

It should be immediately clear, however, that F as defined above will *not* be pseudorandom. Specifically, if x and x' differ only in their first bit then $F_k(x)$ and $F_k(x')$ will differ only in their first byte (regardless of the value of k). In contrast, if F were a truly random permutation then changing the first bit of the input would be expected to affect all bytes of the output.

For this reason, two additional changes are introduced. First, a *diffusion* step is introduced whereby the bits of the output are permuted[4] or "mixed". Second, the confusion/diffusion steps — together called a *round* — are repeated multiple times. As an example, a two-round block cipher would operate as follows: first, $x' := F_k(x)$ would be computed as in Equation (5.1). Then the bits of x' would be re-ordered to give x_1. Then $x'_1 := F_k(x_1)$ would be computed, and the bits of x'_1 would be re-ordered to give the output x_2. We remark that the functions $\{f_i\}$ as well as the permutations used in each round need not be the same. It is typical to assume that the *mixing permutations* used in each round are fixed (i.e., independent of the key), though a dependence on the key could also be introduced.

Repeated use of confusion and diffusion ensures that any small change in the input will be mixed throughout and propagated to all the bits of the output. The effect is that small changes to the input have a significant effect on the output, as one would expect of a random permutation.

[3] Since a random permutation on 8 bits can be represented using $\approx 8 \cdot 2^8$ bits, the length of the key for F is about $16 \cdot 8 \cdot 2^8$ bits, or 32 Kbits. This is much smaller than the $\approx 128 \cdot 2^{128}$ bits that would be required to specify a random permutation on 128 bits.

[4] In this context, "permuting" refers to re-ordering the bits.

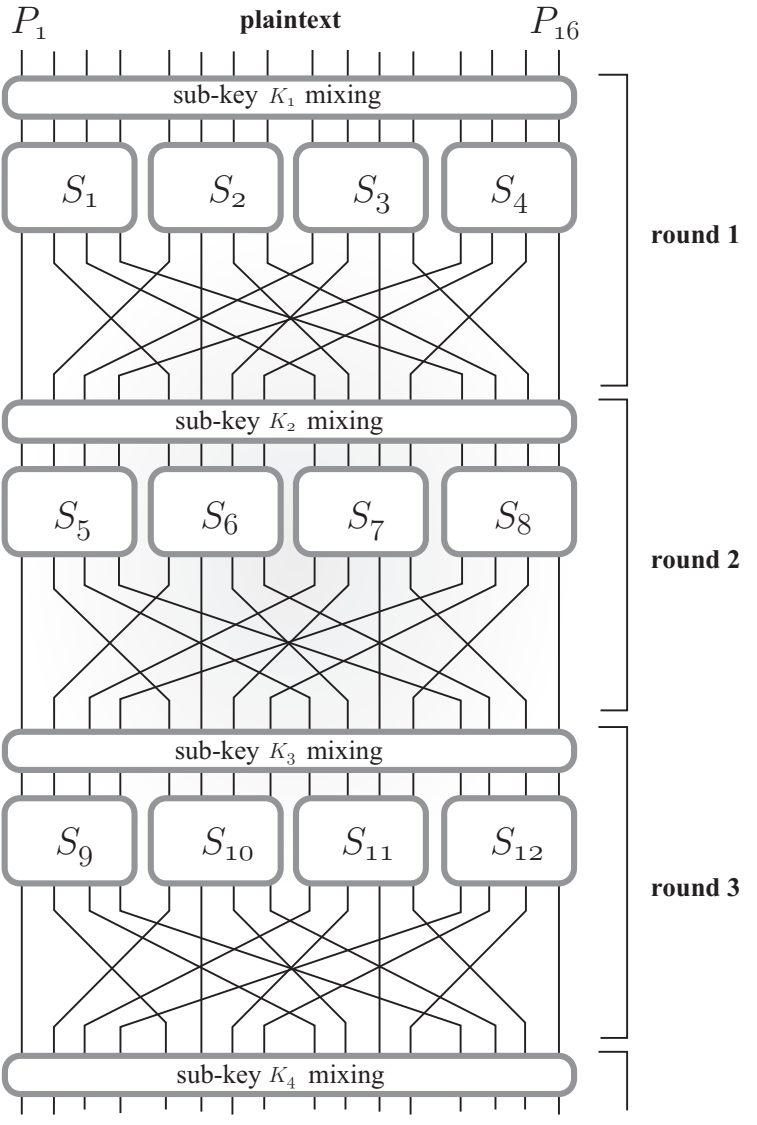

FIGURE 5.1: A substitution-permutation network.

Substitution-permutation networks. A substitution-permutation network can be viewed as a direct implementation of the confusion-diffusion paradigm. The main difference here is that we view the round functions $\{f_i\}$ as being *fixed* (rather than depending on the key k), and the key is used for a different purpose as we will shortly explain. We now refer to the $\{f_i\}$ as *S-boxes* since they act as fixed "substitution functions" (we continue to require that they be permutations).

A substitution-permutation network essentially follows the steps of the confusion-diffusion paradigm outlined earlier. However, since the S-boxes no longer depend on the key, we need to introduce dependence in some other way. (In accordance with Kerckhoffs' principle, we assume that the exact structure of the S-boxes and the mixing permutations are publicly-known, with the only secret being the key.) There are many ways this can be done, but we will focus here on the case where this is done by simply XORing some function of the key with the intermediate results that are fed as input to each round of the network. The key to the block cipher is sometimes referred to as the *master key*, and the sub-keys that are XORed with the intermediate results in each round are derived from the master key according to a *key schedule*. The key schedule is often very simple and may work by just taking subsets of the bits of the key (for example, a two-round network may use the first half of the master key in the first round and the second half of the master key in the second round), though more complex schedules can also be defined. See Figure 5.1 for the high-level structure of a substitution-permutation network, and Figure 5.2 for a closer look at a single round of such a network.

The exact choices of the S-boxes, mixing permutations, and key schedule are what ultimately determine whether a given block cipher is trivially breakable or highly secure. We will only discuss at a cursory level some basic principles behind their design. The first principle is simply a functional requirement, while the second is more specifically related to security.

Design principle 1 — invertibility of the S-boxes. In a substitution-permutation network, the S-boxes must be invertible; that is, they must be one-to-one and onto functions. The reason for this is that otherwise the block cipher will not be a permutation. To see that making the S-boxes one-to-one and onto suffices, we show that when this holds it is possible to fully determine the input given the output and the key. Specifically, we show that every round can be inverted (implying that the entire network can be inverted by working from the end back to the beginning). Recall that a round now consists of three stages: XORing the sub-key with the output of the previous round, passing the result through the S-boxes (as in Equation (5.1)), and finally re-ordering the bits of this result using a mixing permutation. The mixing permutation can easily be inverted since it is just a re-ordering of bits. If the S-boxes are one-to-one and onto, these too can be inverted. The result can then be XORed with the appropriate sub-key to obtain the original input. We therefore have:

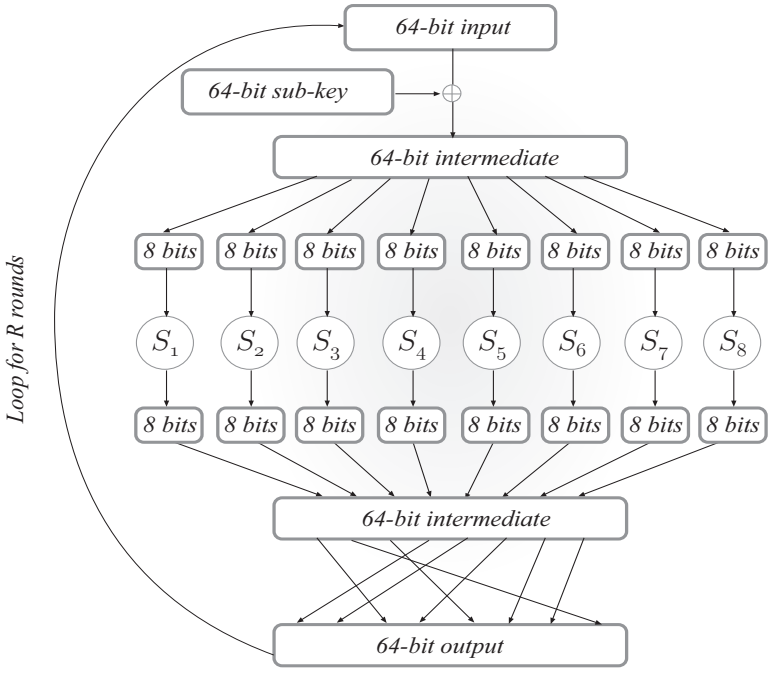

FIGURE 5.2: A single round of a substitution-permutation network.

PROPOSITION 5.1 *Let F be a keyed function defined by a substitution-permutation network in which the S-boxes are all one-to-one and onto. Then regardless of the key schedule and the number of rounds, F_k is a permutation for any choice of k.*

Design principle 2 — the avalanche effect. An important property in any block cipher is that small changes to the input must result in large changes to the output. Otherwise, the outputs of the block cipher on two similar inputs will not look independent (whereas in a random permutation, the outputs of any two unequal inputs are independently distributed). To ensure that this is the case, block ciphers are designed to exhibit the *avalanche effect*, meaning that changing a single bit of the input affects every bit of the output. (This does not mean that changing one bit of the input *changes* every bit of the output, only that it has some effect on every bit of the output. Note that even for a completely random function, changing one bit of the input is expected to change only half the bits of the output, on average.)

It is easy to demonstrate that the avalanche effect holds in a substitution-permutation network provided that the following two properties hold (and sufficiently-many rounds are used):

1. *The S-boxes are designed so that changing a single bit of the input to an S-box changes at least* two bits *in the output of the S-box.*
2. *The mixing permutations are designed so that the output bits of any given S-box are spread into* different *S-boxes in the next round.*

To see how this yields the avalanche effect, assume that the S-boxes are all such that changing a single bit of the input of the S-box results in a change in exactly two bits of the output of the S-box, and that the mixing permutations are chosen as required above. For concreteness, assume the S-boxes have input/output size of 4 bits, and that the block length of the cipher is 128 bits. Consider now what happens when the block cipher is applied to two inputs that differ by only a single bit:

1. After the first round, the intermediate values differ in exactly two bit-positions. This is because XORing the current sub-key maintains the 1-bit difference in the intermediate values, and so the inputs to all the S-boxes except one are identical. In the one S-box where the inputs differ, the output of the S-box causes a 2-bit difference. The mixing permutation applied to the results changes the positions of these differences, but maintains a 2-bit difference.

2. By the second property mentioned earlier, the mixing permutation applied at the end of the first round spreads the two bit-positions where the intermediate results differ into two *different* S-boxes in the second round. (This remains true even after the appropriate sub-key is XORed with the result of the previous round.) So, in the second round there are now *two* S-boxes that receive inputs differing by a single bit. Following the same argument as before, we see that at the end of the second round the intermediate values differ in 4 bits.

3. Continuing with the same argument, we expect 8 bits of the intermediate value to be affected after the 3rd round, 16 bits to be affected after the 4th round, and all 128 bits of the output to be affected at the end of the 7th round.

The last point is not quite precise and it is certainly possible that (depending on the exact inputs, as well as the exact choice of the S-boxes and mixing permutations) there will be fewer differences than expected at the end of some round. For this reason, it is typical to use more than 7 rounds. The importance of the last point is that it gives a *lower bound* on the number of rounds: if fewer than 7 rounds are used then there must be some set of output bits that are not affected, implying that it will be possible to distinguish the cipher from a random permutation.

One might expect that the "best" way to design S-boxes would be to choose them at random (subject to the restriction that they should be one-to-one

and onto). Interestingly, this turns out not to be the case,[5] at least if we want to satisfy the above criterion. For example, consider the case of an S-box operating on 4-bit inputs and let x and x' be two different inputs. Let $y = S(x)$, and now consider choosing $y' \neq y$ at random as the value of $S(x')$. There are 4 strings that differ from y in only 1 bit, and so with probability $4/15 > 1/4$ we will choose y' that does *not* differ from y in two or more bits. The problem is compounded when we consider all inputs, and becomes even worse when we consider that multiple S-boxes are needed. We conclude based on this example that, as a general rule, it is best to carefully design S-boxes with certain desired properties (in addition to the one discussed above) rather than choosing them blindly at random.

In addition to the above, we remark also that randomly-chosen S-boxes are not the best for defending against attacks like the ones we will show in Section 5.6.

Security of Substitution-Permutation Networks

Experience, along with many years of cryptanalytic effort, indicate that substitution-permutation networks are a good choice for constructing pseudorandom permutations as long as great care is taken in the choice of the S-boxes, the mixing permutations, and the key schedule. The Advanced Encryption Standard (AES), described in Section 5.5, is similar in structure to the substitution-permutation network described above, and is widely believed to be a very strong pseudorandom permutation.

It is important to understand, however, that the strength of a cipher constructed in this way depends heavily on the number of rounds used. In order to obtain more of an insight into substitution-permutation networks, we will demonstrate attacks on block ciphers of this type that have very few rounds. These attacks are straightforward, but are worth seeing as they demonstrate conclusively why a large number of rounds are needed.

Attacks on reduced-round substitution-permutation networks. According to the definition of a pseudorandom permutation (see Definition 3.23), the adversary is given an oracle that is either a random permutation or the given block cipher (with a randomly-chosen key). The aim of the adversary is to determine which is the case. Clearly, if an adversary can obtain the secret key of the block cipher, then it can distinguish it from a random permutation. Such an attack is called a *complete break* because once the secret key is learned, no security remains.

[5] The situation here is different from our earlier discussion of the confusion-diffusion paradigm. There, the round functions/S-boxes *depended on the key* and were therefore unknown to the adversary. Here, the round functions are *fixed* and publicly-known, and the question is what fixed functions are best.

Attack on a single-round substitution-permutation network: Let F be a single-round substitution-permutation network. We demonstrate an attack where the adversary is given only a *single* input/output pair (x, y) for a randomly-chosen input value x, and easily learns the secret key k for which $y = F_k(x)$. The adversary begins with the output value y and then inverts the mixing permutation and the S-boxes. It can do this because the specification of the permutation and the S-boxes is public. The intermediate value that the adversary computes from these inversions is exactly $x \oplus k$ (assuming, without loss of generality, that the master key is used as the sub-key in the only round of the network). Since the adversary also has the input x, it immediately derives the secret key k. This is therefore a complete break.

Attack on a two-round substitution-permutation network: We again show an attack that recovers the secret key, though the attack now takes more time. Consider the following concrete parameters. Let the block length of the cipher be 64 bits, and let each S-box have a 4-bit input/output length. Furthermore, let the key k be of length 128 bits where the first half $k^a \in \{0,1\}^{64}$ of the key is used in the first round and the second half $k^b \in \{0,1\}^{64}$ is used in the second round. We use independent keys here to simplify the description of the attack below, but this only makes the attack more difficult.

Say the adversary is given an input x and the output $y = F_k(x)$ of the cipher. The adversary begins by "working backward", inverting the mixing permutation and S-boxes in the second round of the cipher (as in the previous attack). Denote by w_1 the first 4 bits of the result. Letting α_1 denote the first 4 bits of the output of the first round, we have that $w_1 = \alpha_1 \oplus k_1^b$, where k_1^b denotes the first 4 bits of k^b. (The adversary does not know α_1 or k_1^b.) The important observation here is that when "working forward" starting with the input x, the value of α_1 is influenced by at most 4 different S-boxes (because, in the worst case, each bit of α_1 comes from a different S-box in the first round). Furthermore, since the mixing permutation of the first round is known, the adversary knows exactly which of the S-boxes influence α_1. This, in turn, means that at most 16 bits of the key k^a (in known positions) influence the computation of these four S-boxes. It follows that the adversary can guess the appropriate 16 bits of k^a and the 4-bit value k_1^b, and then *verify* possible correctness of this guess using the known input/output pair (x, y). This verification is carried out by XORing the relevant 16 bits of the input x with the relevant 16 bits of k^a, computing the resulting α_1, and then comparing w_1 to $\alpha_1 \oplus k_1^b$ (where k_1^b is also part of the guess). If equality is not obtained, then the guess is certainly incorrect. If equality is obtained, then the guess *may* be correct. Proceeding in this way, the adversary can exhaustively find all values of these 20 bits of the key that are consistent with the given (x, y). This takes time 2^{20} to try each possibility.

If we make the simplifying assumption that an incorrect guess (i.e., one which does not correspond to the bits of the key k that is actually being used) yields a random value for $\alpha_1 \oplus k_1^b$, then we expect an incorrect guess to

pass the above verification test with probability $1/2^{|w_1|} = 1/16$. This means that we expect roughly $2^{20}/16 = 2^{16}$ possibilities to pass the test (including the correct possibility). If we now repeat the above using an additional input/output pair (x', y'), we expect to again eliminate roughly 15/16 of the incorrect possibilities, and we see that if we carry this out repeatedly using many different input/output pairs, we expect to be left with only one guess of the 20 bits of the key that is consistent with all the given input/output pairs. This will, of course, have to be correct.

For concreteness, assume that 8 input/output pairs are used to narrow down to a single possibility. Then the adversary learns the 4 bits of k_1^b in time $8 \cdot 2^{20} = 2^{23}$. This can be repeated for all 16 portions of k^b, leading to an attack with total complexity of $16 \cdot 2^{23} = 2^{27}$ for learning the 64-bit value k^b. Along the way the adversary also learns all 64 bits of k^a. This in fact over-estimates the time complexity of the attack since certain portions of k^a will be re-used, and previously-determined portions of k^a do not need to be guessed again. In any case, an attack having time complexity 2^{27} is well within practical reach, and is much less than the 2^{128} complexity that "should" be required to perform an exhaustive search for a 128-bit key.

There is an important lesson to be learned from the above. The attack is possible since different parts of the key can be isolated from other parts (it is much quicker to carry out 16 attacks of time 2^{23} that reveal 4 bits of k^b each time, than a single attack of time $2^{16 \cdot 4} = 2^{64}$). Thus, further *diffusion* is needed to make sure that all the bits of the key affect all of the bits of the output. Two rounds are not enough for this to take place.

Attack on a three-round substitution-permutation network: In this case we present a weaker attack; instead of learning the key, we just show that it is easy to distinguish a three-round substitution-permutation network from a pseudorandom permutation. This attack is based on the observation, mentioned earlier, that the avalanche effect is not complete after only three rounds (of course, this depends on the block length of the cipher and the input/output length of the S-boxes, but with reasonable parameters this will be the case). Thus, the adversary just needs to ask for the function to be computed on two strings that differ on only a single bit. A three-round block cipher will have the property that many bits of the output will be the same in each case, making it easy to distinguish from a random permutation.

5.2 Feistel Networks

A Feistel network is an alternative approach for constructing a block cipher. The low-level building blocks (S-boxes, mixing permutations, and a key schedule) are the same; the difference is in the high-level design. The

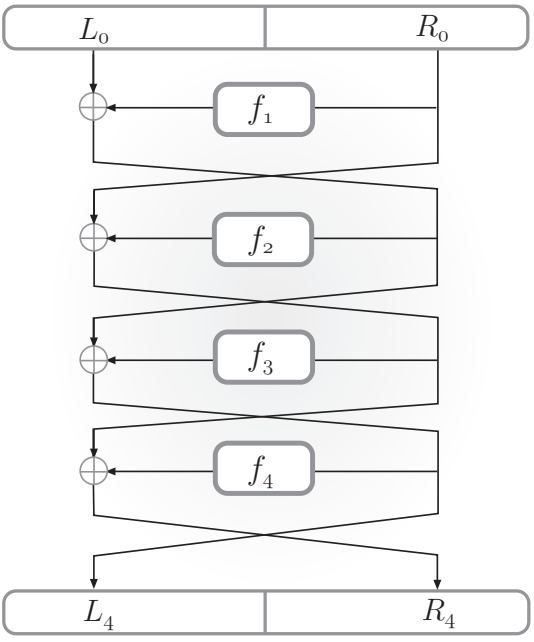

FIGURE 5.3: A 4-round Feistel network.

advantage of Feistel networks over substitution-permutation networks is that a Feistel network eliminates the requirement that S-boxes be invertible. This is important because a good block cipher should have "unstructured" behavior (so that it looks random); however, requiring that all the components of the construction be invertible inherently introduces structure. *A Feistel network is thus a way of constructing an invertible function from non-invertible components.* This seems like a contradiction in terms — if you cannot invert the components, it seems impossible to invert the overall structure — but the Feistel design ingeniously achieves this.

A Feistel network, as in the case of a substitution-permutation network, operates in a series of rounds. In each round, a *round function* is applied in a specific manner that will be described below. In a Feistel network, round functions need not be invertible. Round functions typically contain components like S-boxes and mixing permutations, but a Feistel network can deal with *any* round functions irrespective of their design. When the round functions *are* constructed from S-boxes, the designer has more freedom since the S-boxes need not be invertible.

The ith round of a Feistel network operates as follows. The input to the round is divided into two halves denoted L_{i-1} and R_{i-1} (with L and R denoting the "left half" and "right half" of the input, respectively). If the block length of the cipher is n bits, then L_{i-1} and R_{i-1} each have length $n/2$, and the ith round function f_i will take an $n/2$-bit input and produce an $n/2$-bit

output. (We stress again that although the input and output lengths of f_i are the same, it is not necessarily one-to-one and onto.) The output (L_i, R_i) of the round, where L_i and R_i again denote the left and right halves, is given by

$$L_i := R_{i-1} \quad \text{and} \quad R_i := L_{i-1} \oplus f_i(R_{i-1}). \tag{5.2}$$

In a t-round Feistel network, the n-bit input to the network is parsed as (L_0, R_0), and the output is the n-bit value (L_t, R_t) obtained after applying all t rounds. A 4-round Feistel network is shown in Figure 5.3.

We have not yet discussed how dependence on the key is introduced. As in the case of substitution-permutation networks, the master key k is used to derive sub-keys that are used in each round; the ith round function f_i depends on the ith sub-key, denoted k_i. Formally, the design of a Feistel network specifies a publicly-known *mangler function* $\hat{f}_i$ associated with each round i. This function $\hat{f}_i$ takes as input a sub-key k_i and an $n/2$-bit string and outputs an $n/2$-bit string. When the master key is fixed — thereby fixing each sub-key k_i — the ith round function f_i is defined via $f_i(R) \stackrel{\text{def}}{=} \hat{f}_i(k_i, R)$.

Inverting a Feistel network. A Feistel network is invertible *regardless of the round functions* $\{f_i\}$ (and thus regardless of the mangler functions $\{\hat{f}_i\}$). To show this we need only show that any given round of the network can be inverted. Given the output (L_i, R_i) of the ith round, we can compute (L_{i-1}, R_{i-1}) as follows: first set $R_{i-1} := L_i$. Then compute

$$L_{i-1} := R_i \oplus f_i(R_{i-1}).$$

(The function f_i can be derived from $\hat{f}_i$ if the master key is known.) It can be verified that this gives the correct value (L_{i-1}, R_{i-1}) that was the input of this round (i.e., it computes the inverse of Equation (5.2)). Notice that f_i is evaluated only in the forward direction, as required.

We thus have:

PROPOSITION 5.2 *Let F be a keyed function defined by a Feistel network. Then regardless of the mangler functions $\{\hat{f}_i\}$ and the number of rounds, F_k is a permutation for any choice of k.*

As in the case of substitution-permutation networks, attacks on Feistel networks are possible when the number of rounds is too low. We will see such attacks when we discuss DES in the next section. Theoretical results concerning the security of Feistel networks are discussed in Section 6.6.

5.3 DES – The Data Encryption Standard

The Data Encryption Standard, or DES, was developed in the 1970s at IBM (with help from the National Security Agency), and adopted in 1977 as a Federal Information Processing Standard (FIPS) for the US. In its basic form, DES is no longer considered secure due to its short key length of 56 bits. Nevertheless, it remains in wide use today in its strengthened form of triple-DES (as described in Section 5.4).

DES is of great historical significance, and has undergone intensive scrutiny within the cryptographic community, arguably more than any other cryptographic algorithm in history. The common consensus is that, relative to its key length, DES is extremely secure. Indeed, even after so many years, the best known attack on DES *in practice* is a brute-force search over all 2^{56} possible keys. As we will see later, there are important theoretical attacks on DES that require less computation than such a brute force attack; however, these attacks assume certain conditions that seem unlikely to hold in practice.

In this section, we provide a high-level overview of the main components of DES. We stress that we will not provide a full specification that is correct in every detail, and some parts of the design will be omitted from our description. Our aim is to present the basic ideas underlying the construction of DES, and not all the low-level details; the reader interested in such details can consult the references at the end of this chapter.

5.3.1 The Design of DES

The DES block cipher is a 16-round Feistel network with a block length of 64 bits and a key length of 56 bits. Recall that in a Feistel network the internal f-functions that are used in each round operate on half a block at a time. Thus, the input/output length of a DES round function is 32 bits. The round functions used in each of the 16 rounds of DES are all derived from the same mangler function $\hat{f}_i = \hat{f}$. The *key schedule* of DES is used to derive a 48-bit sub-key k_i for each round from the 56-bit master key k. As discussed in the previous section, the ith round function f_i is then defined as $f_i(R) \stackrel{\text{def}}{=} \hat{f}(k_i, R)$. As is to be expected from the fact that DES uses a Feistel structure, the round functions are *non-invertible*.

The key schedule of DES is relatively simple, with each sub-key k_i being a permuted subset of 48 bits from the master key. We will not describe the key schedule exactly. It suffices for us to note that the 56 bits of the master key are divided into two halves — a "left half" and a "right half" — each containing 28 bits (actually, this division occurs after an initial permutation is applied to the key, but we ignore this in our description). In each round, the left-most 24 bits of the sub-key are taken as some subset of the 28 bits in the left half of the master key, and the right-most 24 bits of the sub-key

are taken as some subset of the 28 bits in the right half of the master key. We stress that the entire key schedule (including the manner in which bits are divided into the left and right halves, and which bits are used in forming sub-key k_i) is fixed and public, and the only secret is the master key itself.

The DES mangler function $\hat{f}$. Recall that the mangler function $\hat{f}$ and the ith sub-key k_i jointly determine the ith round function f_i. The mangler function in DES is constructed using a paradigm we have previously analyzed: it is (essentially) just a 1-round substitution-permutation network! In more detail, computation of $\hat{f}(k_i, R)$ with $k_i \in \{0,1\}^{48}$ and $R \in \{0,1\}^{32}$ proceeds as follows: first, R is *expanded* to a 48-bit value R'. This is done by simply duplicating half the bits of R; we denote this by $R' := E(R)$ where E represents the *expansion function*. Following this step, computation proceeds exactly as in our earlier discussion of substitution-permutation networks: The expanded value R' is XORed with k_i, and the resulting value is divided into 8 blocks, each of which is 6 bits long. Each block is passed through a (different) S-box that takes a 6-bit input and yields a 4-bit output; concatenating the output from the 8 S-boxes gives a 32-bit result. As the final step, a mixing permutation is applied to the bits of this result to obtain the final output of $\hat{f}$. See Figure 5.4 for a diagram of the construction.

One difference here (as compared to our original discussion of substitution-permutation networks) is that the S-boxes referred to above are *not* invertible; indeed, they cannot possibly be invertible since their inputs are longer than their outputs. Further discussion regarding the structural details of the S-boxes is given below.

We stress once again that everything in the above description (including the S-boxes themselves as well as the mixing permutation) is *publicly-known*. The only secret is the master key which is used to derive all the sub-keys.

The S-boxes. The eight S-boxes that form the "core" of $\hat{f}$ are a crucial element of the DES construction, and were very carefully designed (reportedly, with the help of the National Security Agency). Studies of DES have shown that if small changes to the S-boxes had been introduced, or if the S-boxes had been chosen at random, DES would have been much more vulnerable to attack. This should serve as a warning to anyone who wishes to design a block cipher: seemingly arbitrary choices are not arbitrary at all, and if not made correctly may render the entire construction insecure.

Recall that each S-box maps 6-bit strings to 4-bit strings. Each S-box can be viewed as a table with 4 rows and 16 columns, where each cell of the table contains a 4-bit entry. A 6-bit input can be viewed as indexing one of the $2^6 = 64$ cells of the table in the following way: The first and last input bits are used to choose the table row, and bits 2–5 are used to choose the table column. The 4-bit entry at a particular cell represents the output value for the input associated with that position.

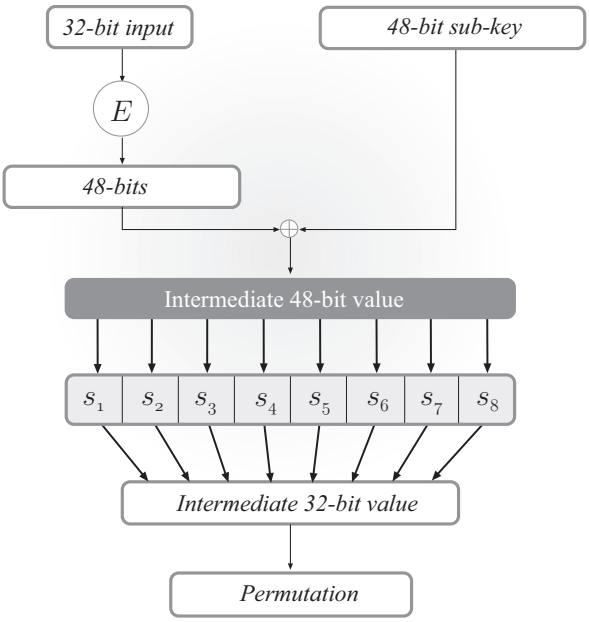

FIGURE 5.4: The DES mangler function.

Due to their importance, we will describe some basic properties of the DES S-boxes:

1. Each S-box is a 4-to-1 function. (That is, exactly 4 inputs are mapped to each possible output.) This follows from the properties below.
2. Each row in the table contains each of the 16 possible 4-bit strings exactly once. (That is, each row is a *permutation* of the 16 possible 4-bit strings.)
3. Changing *one bit* of the input always changes at least *two bits* of the output.

We will use the above properties in our analysis of reduced-round DES below.

The DES avalanche effect. As discussed earlier, the avalanche effect is a crucial property of any secure block cipher. The third property of the DES S-boxes described above, along with the mixing permutation that is used in the mangler function, ensure that DES exhibits a strong avalanche effect. In order to see this, we will trace the difference between the intermediate values in a DES computation of two inputs that differ by just a single bit. Let us denote the two inputs by (L_0, R_0) and (L'_0, R'_0), where we assume that $R_0 = R'_0$ and so the single-bit difference occurs in the left half of the inputs (it may help to refer to Equation (5.2) in what follows). After the first round the intermediate values (L_1, R_1) and (L'_1, R'_1) still differ by only a single bit, though now this

difference is in the right half. In the second round of DES, the right half of each input is run through $\hat{f}$. Assuming that the bit where R_1 and R'_1 differ is not duplicated in the expansion step, the intermediate values before applying the S-boxes still differ by only a single bit. By property 3, the intermediate values *after* the S-box computation differ in at least *two* bits. The result is that the intermediate values (L_2, R_2) and (L'_2, R'_2) differ in *three* bits: there is a 1-bit difference between L_2 and L'_2 (carried over from the difference between R_1 and R'_1) and a 2-bit difference between R_2 and R'_2.

The mixing permutation spreads the two-bit difference between R_2 and R'_2 into different regions of these strings. The effect is that, in the following round, each of the two different bits is used as input for a *different* S-box, resulting in a difference of 4 bits in the right halves of the intermediate values. (There is also now a 2-bit difference in the left halves). As with a substitution-permutation network, we have an exponential effect and so after 7 rounds we expect all 32 bits in the right half to be affected (and after 8 rounds all 32 bits in the left half will be affected as well).

DES has 16 rounds, and so the avalanche effect is completed very early in the computation. This ensures that the computation of DES on similar inputs yields completely different and independent-looking outputs. We remark that the avalanche effect in DES is also due to a careful choice of the mixing permutation, and in fact it has been shown that a *random* mixing permutation would yield a far weaker avalanche effect.

5.3.2 Attacks on Reduced-Round Variants of DES

A useful exercise for understanding more about the DES construction and its security is to look at the behavior of DES with only a few rounds. We will show attacks on one-, two-, and three-round variants of DES (recall that the real DES has 16 rounds). Clearly DES with three rounds or fewer cannot be a pseudorandom function because the avalanche effect is not yet complete after only three rounds. Thus, we will be interested in demonstrating more difficult (and more damaging) *key-recovery attacks* which compute the key k using only a relatively small number of input/output pairs computed using that key. Some of the attacks are similar to those we have seen in the context of substitution-permutation networks; here, however, we will see how they are applied to a concrete block cipher rather than to an abstract design.

All the attacks below will be known-plaintext attacks whereby the adversary has plaintext/ciphertext pairs $\{(x_i, y_i)\}$ with $y_i = DES_k(x_i)$ for some secret key k. When we describe the attacks, we will focus on a particular input/output pair (x, y) and will describe the information about the key that an adversary can derive from this pair. Continuing to use the notation developed earlier, we denote the left and right halves of the input x as L_0 and R_0, respectively, and let L_i, R_i denote the left and right halves after the ith round. We continue to let E denote the DES expansion function, f_i denote the round function applied in round i, and k_i denote the sub-key used in round i.

Single-round DES. In single-round DES, we have that $y = (L_1, R_1)$ where $L_1 = R_0$ and $R_1 = L_0 \oplus f_1(R_0)$. We therefore know an input/output pair for f_1; specifically, we know that $f_1(R_0) = R_1 \oplus L_0$ (note that all these values are known). By applying the inverse of the mixing permutation to the output $R_1 \oplus L_0$, we obtain the intermediate value that contains the outputs from all the S-boxes, where the first 4 bits are the output from the first S-box, the next 4 bits are the output from the second S-box, and so on. This means that we have the exact output of each S-box.

Consider the (known) 4-bit output of the first S-box. Recalling that each S-box is a 4-to-1 function, this means that there are exactly four possible inputs to this S-box that would result in the given output, and similarly for all the other S-boxes; each such input is 6 bits long. The input to the S-boxes is simply the XOR of $E(R_0)$ with the key k_1 used in this round. (Actually, for single-round DES, k_1 is the only key.) Since R_0 is known, we conclude that for each 6-bit portion of k_1 there are four possible values (and we can compute them). This means we have reduced the number of possible keys k_1 from 2^{48} to $4^{48/6} = 4^8 = 2^{16}$ (since there are four possibilities for each of the eight 6-bit portions of k_1). This is already a small number and so we can just try all the possibilities on a different input/output pair (x', y') to find the right one. We thus obtain the full key using only two known plaintexts in time roughly 2^{16}.

Two-round DES. In two-round DES, the output y is equal to (L_2, R_2) where

$$\begin{aligned} L_1 &= R_0 \\ R_1 &= L_0 \oplus f_1(R_0) \\ L_2 &= R_1 = L_0 \oplus f_1(R_0) \\ R_2 &= L_1 \oplus f_2(R_1). \end{aligned}$$

Note that L_0, R_0, L_2, R_2 are known from the given input/output pair (x, y), and thus we also know $L_1 = R_0$ and $R_1 = L_2$. This means that we know the input/output of both f_1 and f_2, and so the same method used in the attack on single-round DES can be used here to determine both k_1 and k_2 in time roughly $2 \cdot 2^{16}$. This attack works even if k_1 and k_2 are completely independent keys. In fact, the key schedule of DES ensures that many of the bits of k_1 and k_2 are equal, which can be used to further speed up the attack.

Three-round DES. See Figure 5.5 for a diagram of three-round DES. The output value y is equal to (L_3, R_3). Since $L_1 = R_0$ and $R_2 = L_3$, the only unknown values in the figure are R_1 and L_2 (which are equal).

Now we no longer have the input/output to any round function f_i. For example, consider f_2. In this case, the output value is equal to $L_1 \oplus R_2$ where both of these values are known. However, we do *not* know the value R_1 that is input to f_2. We do know that $R_1 = L_0 \oplus f_1(R_0)$, and that $R_1 = R_3 \oplus f_3(R_2)$, but the outputs of f_1 and f_3 are unknown. A similar exercise shows that for

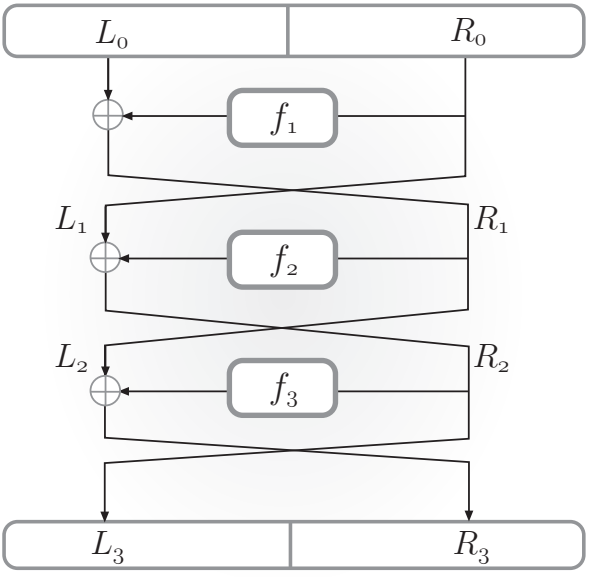

FIGURE 5.5: 3-round DES.

f_1 and f_3 we can determine the inputs but not the outputs. Thus, the attack we used to break one-round and two-round DES will not work here.

Instead of relying on full knowledge of the input and output of one of the round functions, we will use knowledge of a certain relation between the inputs and outputs of f_1 and f_3. Observe that the output of f_1 is equal to $L_0 \oplus R_1 = L_0 \oplus L_2$. The output of f_3 is equal to $L_2 \oplus R_3$. This means that the XOR of the output of f_1 with the output of f_3 is equal to

$$(L_0 \oplus L_2) \oplus (L_2 \oplus R_3) = L_0 \oplus R_3,$$

which is known. That is, *the XOR of the outputs of* f_1 *and* f_3 is known. Furthermore, the input to f_1 is R_0 and the input to f_3 is L_3, both of which are known. We conclude that we can determine the inputs to f_1 and f_3, and the XOR of their outputs. We now describe an attack that finds the secret key based on this information.

Recall that the key schedule of DES has the property that the master key is divided into a "left half", which we denote by k_L, and a "right half" k_R, each containing 28 bits. Furthermore, the left-most bits of the sub-key used in each round are taken only from k_L and the right-most bits of each sub-key are taken only from k_R. This means that the left half of the master key affects the inputs only to the first four S-boxes in any round, while the right half of the master key affects the inputs only to the last four S-boxes. Since the mixing permutation is known, we also know which bits of the output of each round function come out of each S-box.

The idea behind the attack is to separately traverse the key-space for each half of the master key, giving an attack with complexity (roughly) $2 \cdot 2^{28}$ rather than complexity 2^{56}. Such an attack will be possible if we can verify a guess of half the master key, and we now show how this can be done. Let k_L be a guess for the left half of the master key. We know the input R_0 of f_1, and so using our guess of k_L we can compute the input to the first four S-boxes. This means that we can compute half the output bits of f_1 (the mixing permutation spreads out the bits we know, but since the mixing permutation is known we know exactly which bits these are). Likewise, we can compute the same locations in the output of f_3 by using the known input L_3 to f_3 and the same guess k_L. Finally, we can compute the XOR of these output values and check whether they match the appropriate bits in the known value of the XOR of the outputs of f_1 and f_3. If they are not equal, then our guess k_L is incorrect. The correct half-key k_L will always pass this test, and an incorrect half-key is expected to pass this test only with probability roughly 2^{-16} (since we check equality of 16 bits in two computed values). There are 2^{28} possible half-keys to try and so we expect to be left with $2^{28}/2^{16} = 2^{12}$ possibilities for k_L after the above.

By performing the above for each half of the master key, we obtain in time $2 \cdot 2^{28}$ approximately 2^{12} candidates for the left half and 2^{12} candidates for the right half. Since each combination of the left half and right half is possible, we have 2^{24} candidate keys overall and can run a brute-force search over this set using an additional input/output pair (x', y'). The time complexity of the attack is roughly $2 \cdot 2^{28} + 2^{24} < 2^{30}$, and its space complexity is $2 \cdot 2^{12}$. An attack of this complexity could be carried out on a standard personal computer.

5.3.3 The Security of DES

After almost 30 years of intensive study, the best known practical attack on DES is still just an exhaustive search through its key space. (We discuss some important theoretical attacks below. These attacks require a large number of input/output pairs which would be difficult to obtain in an attack on any real-world system using DES.) Unfortunately, the 56-bit key length of DES is short enough that an exhaustive search through all 2^{56} possible keys is now feasible (though still non-trivial). Already in the late '70s there were strong objections to the choice of such a short key for DES. Back then, the objection was theoretical as the computational power needed to search through that many keys was generally unavailable.[6] The practicality of a brute force attack on DES nowadays, however, was demonstrated in 1997 when a number of DES challenges set up by RSA Security were solved (these

[6] In 1977, it was estimated that a computer that could crack DES in one day would cost $20 million to build.

challenges were in the form of input/output pairs and a reward was given to the first person or organization to find the secret key that was used). The first challenge was broken in 1997 by the DESCHALL project using thousands of computers coordinated across the Internet; the computation took 96 days. A second challenge was broken the following year in just 41 days by the `distributed.net` project. A significant breakthrough came later in 1998 when the third challenge was solved in just *56 hours*. This impressive feat was achieved via a special-purpose DES-breaking machine called *Deep Crack* that was built by the Electronic Frontier Foundation at a cost of $250,000. The latest challenge was solved in just over 22 hours (as a combined effort of Deep Crack and distributed.net). The bottom line is that DES has a key that is far too short and cannot be considered secure for any application today.

Less important than the short key length of DES, but still a concern, is its relatively short *block length*. The reason that a small block length is problematic is that the security of many constructions based on block ciphers depends on the block length (*even if the cipher used is "perfect"* and thus regardless of the key length). For example, the proof of security for counter mode encryption (cf. Theorem 3.29) shows that even when a completely random function is used an attacker can break the security of this encryption scheme with probability $\Theta(q^2/2^n)$ if it obtains q plaintext/ciphertext pairs, where n here represents the block length. In the case of DES where $n = 64$, this means that if an attacker obtains only $q = 2^{27}$ plaintext/ciphertext pairs, security is compromised with high probability. Obtaining plaintext/ciphertext pairs is relatively easy if an adversary eavesdrops on the encryption of messages containing known headers, redundancies, etc. (though obtaining 2^{27} such pairs may be out of reach).

We stress that the insecurity of DES has nothing to do with its internal structure and design, but rather is due only to its short key length (and, to a lesser extent, its short block length). This is a great tribute to the designers of DES who seem to have succeeded in constructing an almost "perfect" block cipher (with the glaring exception of its too-short key[7]). Since DES itself seems not to have significant structural weaknesses, it makes sense to use DES as a building block in order to construct a block cipher with a longer key. We discuss such an approach in Section 5.4.

Looking ahead a bit, we note that the Advanced Encryption Standard (AES) — the replacement for DES — was explicitly designed to address concerns regarding the short key length and block length of DES. AES supports keys of length 128 bits (and more), and a block length of 128 bits.

[7] Actually, the designers of DES almost certainly recognized that the key was too short; it has been suggested that the NSA requested that the key be short enough for them to crack.

Advanced Cryptanalytic Attacks on DES

The successful brute-force attacks described above do not utilize any internal weaknesses of DES. Indeed, for many years no such weaknesses were known to exist. The first breakthrough on this front was by Biham and Shamir in the late '80s who developed a technique called *differential cryptanalysis* and used it to design an attack on DES using less time than a brute-force search. Their specific attack takes time 2^{37} (and uses negligible memory) but requires the attacker to analyze 2^{36} ciphertexts obtained from a pool of 2^{47} chosen plaintexts. While the existence of this attack was a breakthrough result from a theoretical standpoint, it does not appear to be of much practical concern since it is hard to imagine any realistic scenario where an adversary can obtain this many values in a chosen-plaintext attack.

Interestingly, the work of Biham and Shamir indicated that the DES S-boxes had been specifically designed to be resistant to differential cryptanalysis (to some extent), suggesting that the technique of differential cryptanalysis was known (but not publicly revealed) by the designers of DES. After Biham and Shamir announced their result, the designers of DES claimed that they were indeed aware of differential cryptanalysis and had designed DES to thwart this type of attack (but were asked by the NSA to keep it quiet in the interests of national security).

Following Biham and Shamir's breakthrough, *linear cryptanalysis* was developed by Matsui in the early '90s and was also applied successfully to DES. The advantage of Matsui's attack is that although it still requires a large number of outputs (2^{43} to be exact), they may be arbitrary and need not be chosen by the attacker. (That is, it utilizes a known-plaintext attack rather than a chosen-plaintext attack.) Nevertheless, it is still hard to conceive of any real scenario where it would be possible to obtain such a large number of input/output (or plaintext/ciphertext) pairs.

We briefly describe the basic ideas behind differential and linear cryptanalysis in Section 5.6. In conclusion, however, we emphasize that although it is possible to break DES in less time than that required by a brute-force search using sophisticated cryptanalytic techniques, an exhaustive key search is still the most effective attack in practice.

5.4 Increasing the Key Length of a Block Cipher

The only known practical weakness of DES is its relatively short key. It thus makes sense to try to design a block cipher with a larger key length using "basic" DES as a building block. Some approaches to doing so are discussed in this section. Although we refer to DES throughout the discussion, and DES is the most prominent instance where these techniques have been applied, everything we say here applies generically to any block cipher.

Internal tampering vs. black-box constructions. There are two general approaches one could take to constructing another cipher based on DES. The first approach would be to somehow modify the *internal structure* of DES, while increasing the key length. For example, one could leave the mangler function untouched and simply use a 128-bit master key with a different key schedule (still choosing a 48-bit sub-key in each round). Or, one could change the S-boxes themselves and use a larger sub-key in each round. The disadvantage of this approach is that by modifying DES — in even the smallest way — we lose the confidence we have gained in DES by virtue of the fact that it has remained secure for so many years. More to the point, cryptographic constructions are very sensitive and thus even mild, seemingly-insignificant changes can render the original construction completely insecure.[8] Changing the internals of a block cipher is therefore not recommended. (If it is really deemed necessary to introduce changes in a block cipher, it is usually better to just develop a new cipher from scratch. This, of course, is all predicated on the assumption that this will be done by experts — we do not recommend ever designing a new cipher especially when modern block ciphers like AES are available.)

An alternative approach that does not suffer from the above problem is to use DES as a "black box". That is, in this approach we completely ignore the internal structure of DES and treat it as a black box that implements a "perfect" block cipher with a 56-bit key. Then, a new cipher is constructed that uses only invocations of the original unmodified DES. Since DES itself is not changed at all, this approach is much more likely to lead to a secure cipher (though it may also lead to a less efficient one), and is the approach we will explore here.

Double Encryption

Let F be a block cipher. Then a new block cipher F' with a key that is twice the length of the original one can be defined by

$$F'_{k_1,k_2}(x) \stackrel{\text{def}}{=} F_{k_2}(F_{k_1}(x)),$$

where k_1 and k_2 are independent keys. If F is DES then the result is a block cipher F' taking a 112-bit key. If exhaustive key search were the best available attack on F', a key length of 112 bits would be sufficient since an attack requiring time 2^{112} is completely out of reach. Unfortunately, we now show an attack on F' that runs in time roughly 2^n when the original keys k_1 and k_2 are each of length n (and the block length is at least n); this is significantly less than the 2^{2n} time one would hope would be necessary to carry out an exhaustive search for a $2n$-bit key. This means that the new

[8] In fact, various results to this effect have been shown for DES; e.g., changing the S-boxes very slightly makes DES much more vulnerable to attack.

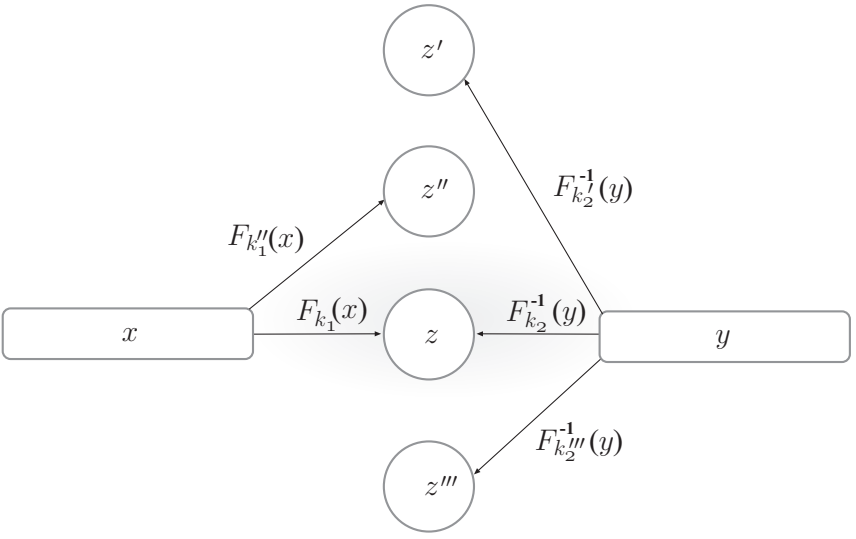

FIGURE 5.6: A meet-in-the-middle attack.

block cipher is essentially no better than the old one, even though it has a key that is twice as long.[9]

The attack is called a "meet-in-the-middle attack" for reasons that will soon become clear. Say the adversary is given a single input/output pair (x, y) where $y = F'_{k_1,k_2}(x) = F_{k_2}(F_{k_1}(x))$. The adversary will narrow down the set of possible keys in the following way:

1. Set $S := \emptyset$.
2. For each $k_1 \in \{0,1\}^n$, compute $z := F_{k_1}(x)$ and store (z, k_1) in a list L.
3. For each $k_2 \in \{0,1\}^n$, compute $z := F^{-1}_{k_2}(y)$ and store (z, k_2) in a list L'.
4. Sort L and L', respectively, by their first components.
5. Say that an entry $(z_1, \tilde{k}_1)$ in L and another entry $(z_2, \tilde{k}_2)$ in L' are a *match* if $z_1 = z_2$. For each match of this sort, add $(\tilde{k}_1, \tilde{k}_2)$ to S.

The set S output by this algorithm contains exactly those values (k_1, k_2) for which $y = F'_{k_1,k_2}(x)$. This holds because it outputs exactly those values (k_1, k_2) satisfying

$$F_{k_1}(x) = F^{-1}_{k_2}(y), \tag{5.3}$$

which holds if and only if $y = F'_{k_1,k_2}(x)$. See Figure 5.6 for a graphical depiction of the attack.

[9] This is not quite true since a brute-force attack on F can be carried out in time 2^n and constant memory, whereas the attack on F' requires 2^n time *and* 2^n *memory* (and memory is more precious than time). Nevertheless, the attack illustrates that F' does not achieve a sufficient level of security.

If n is also the block length of F then a random pair (k_1, k_2) is expected to satisfy Equation (5.3) with probability roughly 2^{-n}, and so the number of elements in S will be approximately $2^{2n}/2^n = 2^n$. Given another two input/output pairs and trying all 2^n elements of S with respect to these pairs is expected to identify the correct (k_1, k_2) with very high probability.

Complexity. The lists L and L' can be generated and sorted (using, e.g., counting sort) in time roughly 2^n. Once the lists are sorted, all matches can be found in time $\mathcal{O}(|S| + |L| + |L'|) = \mathcal{O}(2^n)$ (see Exercise 5.13). Determining the correct key using an additional pair (x', y') takes time $\mathcal{O}(|S|) = 2^n$. We conclude that the time complexity of the above algorithm is $\mathcal{O}(2^n)$.

Triple Encryption

The obvious generalization of the preceding approach is to apply the block cipher three times in succession. Two variants of this approach are common:

1. *Variant 1 — three independent keys:* Choose 3 independent keys k_1, k_2, k_3 and define $F'_{k_1,k_2,k_3}(x) \stackrel{\text{def}}{=} F_{k_3}(F^{-1}_{k_2}(F_{k_1}(x)))$.
2. *Variant 2 — two independent keys:* Choose 2 independent keys k_1, k_2 and define $F'_{k_1,k_2}(x) \stackrel{\text{def}}{=} F_{k_1}(F^{-1}_{k_2}(F_{k_1}(x)))$.

Before comparing the security of the two alternatives we note that the middle invocation of the original cipher is actually in the reverse direction. If F is a sufficiently good cipher this makes no difference to the security, since if F is a strong pseudorandom permutation then F^{-1} must be too. The reason for this strange alternation between F, F^{-1}, and F is so that if one chooses $k_1 = k_2 = k_3$, the result is a single invocation of F with k_1. This ensures backward compatibility (i.e., in order to switch back to a single invocation of F, it suffices to just set the keys to all be equal).

Security of the first variant. The key length of this variant is $3n$ (where, as before, the key length of the original cipher F is n) and so we might hope that the best attack on this cipher would require time 2^{3n}. However, the cipher is susceptible to a meet-in-the-middle attack just as in the case of double encryption, though the attack now takes time 2^{2n}. This is the best known attack. Thus, although this variant is not as secure as we might have hoped, it obtains sufficient security for all practical purposes if $n = 56$ (assuming, of course, that the original block cipher F has no weaknesses).

Security of the second variant. The key length of this variant is $2n$ and so the best we can hope for is security against attacks running in time 2^{2n}. There is no known attack with better time complexity when the adversary is given only a single input/output pair. However, there is a known chosen-plaintext attack that finds the key in time 2^n using 2^n chosen input/output pairs (see Exercise 5.12). Despite this, it is still a reasonable choice in practice.

Triple-DES (3DES). Triple-DES is based on a triple invocation of DES using two or three keys, as described above. It is widely believed to be highly secure and in 1999 officially replaced DES as a standard. Triple-DES is still widely used today and is considered a very strong block cipher. Its only drawbacks are its relatively small block length and the fact that it is quite slow since it requires 3 full block cipher operations. These drawbacks have led to the replacement of DES/triple-DES by the Advanced Encryption Standard (AES), presented in the next section.

5.5 AES – The Advanced Encryption Standard

In January 1997, the United States National Institute of Standards and Technology (NIST) announced that it would hold a competition to select a new block cipher — to be called the *Advanced Encryption Standard*, or AES — to replace DES. The competition began with an open call for teams to submit candidate block ciphers for evaluation. A total of 15 different algorithms were submitted from all over the world, and these submissions included the work of many of the best cryptographers and cryptanalysts today. Each team's candidate cipher was intensively analyzed by members of NIST, the public, and (especially) the other teams. Two workshops were held, one in 1998 and one in 1999, at which cryptanalytic attacks of the various submissions were shown. Following the second workshop, NIST narrowed the field down to 5 "finalists" and the second round of the competition began. A third AES workshop was held, inviting additional scrutiny on the five finalists. In October 2000, NIST announced that the winning algorithm was Rijndael (a block cipher designed by John Daemen and Vincent Rijmen from Belgium), though it conceded that any of the 5 finalists would have made an excellent choice. In particular, no serious security vulnerabilities were found in any of the 5 finalists, and the selection of a "winner" was based in part on properties such as efficiency, flexibility, etc.

The process of selecting AES was ingenious because any group who submitted an algorithm, and was therefore interested in having their algorithm adopted, had strong motivation to find attacks in all the other submissions.[10] In this way, essentially all the world's best cryptanalysts focused on finding even the slightest weaknesses in the candidate ciphers submitted to the competition. After only a few years each candidate algorithm was already subjected to intensive study, thus increasing our confidence in the security of the winning algorithm. Of course, the longer the algorithm is used without

[10] The motivation was not financial because the winning submission could not be patented. Nevertheless, much honor and glory was at stake.

being broken, the more our confidence will grow. Today, AES is already very widely used and no significant security weaknesses have been discovered.

The AES construction. In this section, we will present the high-level structure of Rijndael/AES. (Technically speaking, Rijndael and AES are not the same thing but the differences are unimportant for our discussion here.) As with DES, we will not present a full specification and our description should not be used as a basis for implementation. Our aim is only to provide a general idea of how the algorithm works.

The AES block cipher has a 128-bit block length and can use 128-, 192-, or 256-bit keys. The length of the key affects the key schedule (i.e., the sub-key that is used in each round) as well as the number of rounds, but does not affect the high-level structure of each round.

In contrast to DES that uses a Feistel structure, AES is essentially a substitution-permutation network. During computation of the AES algorithm, a 4-by-4 array of bytes called the *state* is modified in a series of rounds. The state is initially set equal to the input to the cipher (note that the input is 128 bits which is exactly 16 bytes). The following operations are then applied to the state in a series of four stages during each round:

1. *Stage 1* — AddRoundKey: In every round of AES, a 128-bit sub-key is derived from the master key, and is interpreted as a 4-by-4 array of bytes. The state array is updated by XORing it with this sub-key.

2. *Stage 2* — SubBytes: In this step, each byte of the state array is replaced by another byte according to a single fixed lookup table S. This substitution table (or S-box) is a bijection over $\{0,1\}^8$. We stress that there is only *one* S-box and it is used for substituting *all* the bytes in the state array, in every round.

3. *Stage 3* — ShiftRows: In this step, the bytes in each row of the state array are cyclically shifted to the left as follows: the first row of the array is untouched, the second row is shifted one place to the left, the third row is shifted two places to the left, and the fourth row is shifted three places to the left. All shifts are cyclic so that, e.g., in the second row the first byte becomes the fourth byte.

4. *Stage 4* — MixColumns: In this step, an invertible linear transformation is applied to each column. One can think of this as matrix multiplication (over some appropriate field).

By viewing stages 3 and 4 together as a "mixing" step, we see that each round of AES has the structure of a substitution-permutation network: the round sub-key is first XORed with the input to the current round; next, a small, invertible function is applied to "chunks" of the resulting value; finally, the bits of the result are mixed in order to obtain diffusion. The only difference is that, unlike our general description of substitution-permutation networks,

here the mixing step does not consist of a simple permutation of the bits but is instead carried out using an invertible linear transformation of the bits. (Simplifying things a bit and looking at a trivial 3-bit example, a permutation of the bits of $x = x_1 \| x_2 \| x_3$ might, e.g., map x to $x' = x_2 \| x_1 \| x_3$. An invertible linear transformation might map x to $x_1 \oplus x_2 \| x_2 \oplus x_3 \| x_1 \oplus x_2 \oplus x_3$.)

The number of rounds in AES depends on the length of the key. There are 10 rounds for a 128-bit key, 12 rounds for a 192-bit key, and 14 rounds for a 256-bit key. In the final round of AES the MixColumns stage is replaced with an additional AddRoundKey step (this prevents an adversary from simply inverting the last three stages, which do not depend on the key).

Security of AES. As we have mentioned, the AES cipher was subject to intensive scrutiny during the selection process and this has continued ever since. To date, the only non-trivial cryptanalytic attacks that have been found are for reduced-round variants of AES. It is often hard to compare cryptanalytic attacks because each tends to perform better with regard to some parameter; we describe the complexity of one set of attacks merely to give a flavor of what is known. There are known attacks on 6-round AES for 128-bit keys (using on the order of 2^{72} encryptions), 8-round AES for 192-bit keys (using on the order of 2^{188} encryptions), and 8-round AES for 256-bit keys (using on the order of 2^{204} encryptions). We stress that the above attacks are for *reduced-round* variants of AES, and as of today no attack better than exhaustive key search is known for the full AES construction. (Moreover, even the complexities of the attacks on the reduced-round variants are very high.)

We conclude that, as of today, AES constitutes an excellent choice for almost any cryptographic implementation that relies on a pseudorandom permutation. It is free, standardized, efficient, and highly secure.

5.6 Differential and Linear Cryptanalysis – A Brief Look

Typical block ciphers are relatively complicated constructions, and as such are hard to analyze and cryptanalyze. Nevertheless, one should not be fooled into thinking that a complicated cipher is difficult to break. On the contrary, it is very difficult to construct a secure block cipher and surprisingly easy to construct a trivially insecure one (no matter how complicated it looks). This should serve as a warning that non-experts should not try to construct new ciphers unless there is a very good reason. Given the availability of triple-DES and AES, in most applications it is hard to justify using anything else.

In this section we will briefly mention two tools that are now a standard part of the cryptanalyst's toolbox. The existence of such tools should also serve to reinforce the above warning that it is very hard to construct good block ciphers.

Differential cryptanalysis. This technique was first presented in the late '80s by Biham and Shamir, who used it to attack DES in 1993. The basic idea behind the attack is to tabulate *specific differences in the input* that lead to *specific differences in the output* with probability greater than would be expected for a random permutation. Specifically, say a block cipher has block length n and let $\Delta_x, \Delta_y \in \{0,1\}^n$. We say that *the differential* (Δ_x, Δ_y) *appears with probability* p if for random inputs x_1 and x_2 satisfying $x_1 \oplus x_2 = \Delta_x$ and random choice of key k, the probability that $F_k(x_1) \oplus F_k(x_2) = \Delta_y$ is p. It is clear that for a random function, no differential should appear with probability much higher than 2^{-n}. In a weak block cipher, however, there may be differentials that appear with significantly higher probability.

One can find a differential that occurs with high probability either through a brute-force search (done once-and-for-all for the block cipher, independent of any particular key) or via a careful analysis of the block cipher itself. If a differential exists with probability $p \gg 2^{-n}$ then the block cipher already no longer qualifies as a pseudorandom permutation. The point of differential cryptanalysis is to use many differentials, that may each be only slightly larger than 2^{-n}, to recover the secret key (and thus break the cipher entirely) using a chosen-plaintext attack. We will not discuss the details here, though we mention that applying the block cipher to random pairs of inputs that have the given differential enables a cryptanalyst to isolate portions of the secret key and verify guesses for those portions. As we discussed regarding the attack on a 2-round substitution-permutation network, the ability to isolate parts of a key enables an attacker to obtain the key in time less than a brute force search. Note, however, that the fact that chosen plaintexts are required makes the attack of somewhat limited practicality.

Although differential cryptanalysis does not appear to lead to any practical attacks on DES and AES (since the number of chosen plaintexts required to carry out the attack is huge), differential cryptanalysis has been used successfully to attack other block ciphers. One important example is FEAL-8, which was completely broken using differential cryptanalysis.

Linear cryptanalysis. Linear cryptanalysis was developed by Matsui in the early '90s. This method considers linear relationships between the input and output. Say that bit positions $i_1, \ldots, i_\ell$ and $i'_1, \ldots, i'_\ell$ have *bias* p if, for randomly-chosen input x and key k, it holds that

$$\Pr[x_{i_1} \oplus \cdots \oplus x_{i_\ell} \oplus y_{i'_1} \oplus \cdots \oplus y_{i'_\ell} = 0] = p,$$

where $y = F_k(x)$ and x_i, y_i represent the bits of x and y. For a truly random function, we expect the bias to be close to 0.5. Matsui showed how to use a large enough bias in a given cipher F to completely break the cipher by finding the secret key. An important feature of this attack is that it does not require *chosen* plaintexts, and *known* plaintexts are sufficient. Even so, if a very large number of plaintext/ciphertext pairs are needed the attack becomes impractical in most settings.

Additional Reading and References

The confusion-diffusion paradigm and substitution-permutation networks were introduced by Shannon [127] and Feistel [53]. See the thesis of Heys [78] for further information regarding the design of substitution-permutation networks. Feistel networks were first described in [53]. A theoretical analysis of them was given by Luby and Rackoff [97], and will be discussed in Chapter 6.

The DES standard can be found at [109], and a more reader-friendly description can be found in the textbook by Kaufman et al. [87]. Details of the competition leading to the selection of Rijndael as the AES can be found at `http://csrc.nist.gov/CryptoToolkit/aes/index.html`. A comprehensive description of AES can be found in [87] as well as the book written by its designers, Daemen and Rijmen [41]. Cid et al. show an approach that may lead to cryptanalytic attacks on AES [33]. There are a large number of other good (and less good) block ciphers in the literature. For a broad but somewhat outdated overview of other ciphers, see [99, Chapter 7].

The meet-in-the-middle attack on double encryption is due to Diffie and Hellman [48]. The attack on two-key triple encryption mentioned in the text (and explored in Exercise 5.12) is by Merkle and Hellman [103]. Theoretical analysis of the security of double and triple encryption can be found in [3, 17].

DESX is another technique for increasing the effective key length of DES, without using additional invocations of DES. The secret key consists of the values $k_i, k_o \in \{0,1\}^{64}$ and $k \in \{0,1\}^{56}$, and the cipher is defined by

$$DESX_{k_i,k,k_o}(x) = k_o \oplus DES_k(x \oplus k_i).$$

This methodology was first studied by Even and Mansour [52]. Its concrete application to DES was proposed by Rivest, and its security was later analyzed by Kilian and Rogaway [88].

Differential cryptanalysis was introduced by Biham and Shamir [18] and its application to DES is described in [19]. Coppersmith [34] describes the DES design in light of the public discovery of differential cryptanalysis. Linear cryptanalysis was discovered by Matsui [98]. Langford's thesis [93] contains further improvements of differential and linear cryptanalysis, and also surveys known attacks on DES (and reduced-round variants) as of 1995. For more information on these advanced cryptanalytic techniques, we refer the reader to the excellent tutorial on differential and linear cryptanalysis by Heys [77]. A more concise presentation can be found in the textbook by Stinson [138].

Exercises

5.1 Say a block cipher F has the property that, given oracle access to F_k for randomly-chosen key k, it is possible to determine k using only 100

queries to the oracle (and minimal computation time). Show formally that F cannot be a pseudorandom permutation.

5.2 In our attack on a two-round substitution-permutation network, we considered a block length of 64 bits and a network with 16 S-boxes that each take a 4-bit input. Repeat the analysis for the case of 8 S-boxes, each taking an 8-bit input. What is the complexity of the attack now? Repeat the analysis again with a 128-bit block length and 16 S-boxes that each take an 8-bit input. Does the block length make any difference?

5.3 Our attack on a three-round substitution-permutation network does not recover the key but only shows how to distinguish the cipher from a random permutation. Thus it is not a "complete break". Despite this, show that using a three-round substitution-permutation network in the *counter-mode* encryption scheme (see Section 3.6.4) can have disastrous effects on the security of encryption.

5.4 Consider a modified substitution-permutation network where instead of carrying out the key-mixing, substitution, and permutation steps in alternating order for r rounds, the cipher instead first applies r rounds of key-mixing, then carries out r rounds of substitution, and finally applies r permutations. Analyze the security of this construction.

5.5 What is the output of an r-round Feistel network when the input is (L_0, R_0) in each of the following two cases:

(a) Each round function outputs all 0s, regardless of the input.

(b) Each round function is the identity function.

5.6 Show that DES has the property that $DES_k(x) = \overline{DES_{\overline{k}}(\overline{x})}$ for every key k and input x (where $\overline{z}$ denotes the bitwise complement of z). This is called the *complementarity property* of DES. (The description of DES given in this chapter is sufficient for this exercise.)

5.7 Use the previous exercise to show how it is possible to find the secret key in DES (with probability 1) in time 2^{55}.

Hint: Use a chosen-plaintext attack with two carefully chosen plaintexts.

5.8 In the actual construction of DES, the two halves of the output of the final round of the Feistel network are swapped. That is, if the output of the final round is (L_{16}, R_{16}) then the output of the cipher is in fact (R_{16}, L_{16}). Show that the only difference between the computation of DES_k and DES_k^{-1} (given the swapping of halves) is the order of subkeys.

5.9 (This exercise assumes the results of the previous exercise.)

(a) Show that for $k = 0^{56}$ it holds that $DES_k(DES_k(x)) = x$. Why does the use of such a key pose a security threat?

(b) Find three other DES keys with the same property. These keys are known as *weak keys* for DES.

(c) Does the existence of these 4 weak keys represent a serious vulnerability in DES? Explain your answer.

5.10 Describe attacks on the following modifications to DES:

(a) Each round sub-key is 32 bits long, and the mangler function simply XORs the round sub-key with the input to the round (i.e., $\hat{f}(k, R) = k_i \oplus R$). For this example, the key schedule is unimportant and you can treat the k_i as independent keys.

(b) Instead of using different sub-keys in every round, the same 48-bit sub-key is used in every round. Show how to distinguish the cipher from a random permutation without a 2^{48}-time brute-force search.

Hint: Exercises 5.8 and 5.9 may help...

5.11 Show an improvement to the attack on three-round DES that recovers the key using two input/output pairs but runs in time $2 \cdot 2^{28} + 2 \cdot 2^{12}$.

5.12 This question illustrates an attack on two-key triple encryption. Let F be a block cipher with n-bit block length and key length, and set $F'_{k_1,k_2}(x) = F_{k_1}(F^{-1}_{k_2}(F_{k_1}(x)))$.

(a) Assume that given a pair (m_1, m_2) it is possible to find in *constant* time all keys k_2 such that $m_2 = F^{-1}_{k_2}(m_1)$. Show how to recover the entire key for F' (with high probability) in time roughly 2^n using three known input/output pairs.

(b) In general, it will *not* be possible to find k_2 as above in constant time. However, show that by using a pre-processing step taking 2^n time it is possible, given m_2, to find in (essentially) constant time all keys k_2 such that $m_2 = F^{-1}_{k_2}(0^n)$.

(c) Assume k_1 is known and that the pre-processing step above has already been run. Show how to use a single pair (x, y) for a *chosen* input value x to determine k_2 in constant time.

(d) Put the above components together to devise an attack that recovers the entire key by running in roughly 2^n time and requesting the encryption of roughly 2^n chosen inputs.

5.13 Show how all matches can be found in the meet-in-the-middle attack on double encryption in time *linear* in the number of matches and the lengths of the two sorted lists.

5.14 Say the key schedule of DES is modified as follows: the left half of the master key is used to derive all the sub-keys in rounds 1–8, while the right half of the master key is used to derive all the sub-keys in rounds 9–16. Show an attack on this modified scheme that recovers the entire key in time roughly 2^{28}.

5.15 Consider using DES as a fixed-length collision-resistant hash function in the following way: Define $h : \{0,1\}^{112} \to \{0,1\}^{64}$ as $h(x_1 \| x_2) \stackrel{\text{def}}{=} DES_{x_1}(DES_{x_2}(0^{64}))$ where $|x_1| = |x_2| = 56$.

(a) Write down an explicit collision in h.

Hint: Use Exercise 5.9.

(b) Show how to find a pre-image of a given value y (that is, x_1, x_2 such that $h(x_1 \| x_2) = y$) in roughly 2^{56} time.

(c) Show a more clever pre-image attack that runs in roughly 2^{32} time and succeeds with high probability.

Hint: Rely on the results of Appendix A.4.

Chapter 6

* Theoretical Constructions of Pseudorandom Objects

In Chapter 3 we introduced the notion of pseudorandomness, and defined the basic cryptographic primitives of pseudorandom generators, functions, and permutations. We showed in Chapters 3 and 4 that these primitives serve as the basic building blocks for all of private-key cryptography. As such, it is of great importance to understand these primitives from a theoretical point of view. In this chapter we formally introduce the concept of *one-way functions* — functions that are, informally speaking, easy to compute but hard to invert — and show how pseudorandom generators and pseudorandom permutations can be constructed under the sole assumption that one-way functions exist.[1] Moreover, we will see that one-way functions are a necessary assumption for essentially any non-trivial cryptographic task in the private-key setting. Tying everything together, this means that *the existence of one-way functions is equivalent to the existence of all (non-trivial) private-key cryptography.* This result constitutes one of the major contributions of modern cryptography.

The constructions of, say, pseudorandom permutations based on one-way functions that we show in this chapter should be viewed as complementary to the constructions of block ciphers given in the previous chapter. The focus of the previous chapter was on how pseudorandom permutations are currently realized in practice, and the intent of that chapter was to introduce some basic approaches and design principles that are used in their construction. Somewhat disappointing, however, was the fact that none of the constructions we showed could be *proven* secure based on any weaker (i.e., more reasonable) assumptions. In contrast, in the present chapter we will prove that it is possible to construct pseudorandom permutations starting from the very mild assumption that one-way functions exist. This assumption is more reasonable than assuming, say, that DES is a pseudorandom permutation since we have a number of candidate one-way functions that have been studied for many years, even before the advent of cryptography. (See the very beginning of Chapter 5 for further discussion of this point.) The downside, though, is that the constructions we show here are all far less efficient than those of Chapter 5, and thus are not actually used. It remains an important challenge for cryp-

[1] Actually, this is not quite true since we are for the most part going to rely on one-way *permutations* in this chapter. Nevertheless, it is known that one-way functions suffice.

tographers to "bridge this gap" and develop provably-secure constructions of pseudorandom generators, functions and permutations whose efficiency is comparable to the best available stream and block ciphers.

A note regarding this chapter. The material in this chapter is somewhat more advanced than the other material in this book. As indicated by the fact that the chapter is "starred", the material presented here is not used in the remainder of the book and so this chapter can be skipped if desired. Having said this, we have tried to present the material in such a way that it is understandable (with effort) to an advanced undergraduate or beginning graduate student. We highly encourage everyone to read Sections 6.1 and 6.2 which introduce one-way functions and provide an overview of the constructions described in the rest of this chapter. We believe that familiarity with at least some of the topics covered here is important enough to warrant the effort involved.

6.1 One-Way Functions

In this section we formally define one-way functions, and then briefly discuss some candidates that are widely believed to satisfy this definition. (We will see many more examples of conjectured one-way functions in Chapters 7 and 11.) We conclude this section by introducing the notion of *hard-core predicates*; these can be viewed (in some sense) as "encapsulating" the hardness of inverting a one-way function, and will be used extensively in the constructions that follow in subsequent sections.

6.1.1 Definitions

A one-way function f has the property that it is easy to compute, but hard to invert. The first condition is easy to formalize: we will simply require that f be computable in polynomial time. As for the second condition, since we are ultimately interested in building cryptographic schemes that are hard for a probabilistic polynomial-time adversary to break except with negligible probability, we will formalize the hardness of inverting f by requiring that it be infeasible for any probabilistic polynomial-time algorithm to invert f — that is, to find a pre-image of a given value y — except with negligible probability. (It is always possible to find a pre-image with negligible probability just by guessing. Likewise, it is always possible to find a pre-image in exponential time by performing a brute-force search over the domain of f.) An important technical point is that this probability is taken over an experiment in which y is generated by choosing an element x of the domain at random and then setting $y := f(x)$ (rather than choosing y at random from the range). This will become clear from the formal definition that follows.

Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. Consider the following experiment defined for any algorithm $\mathcal{A}$ and any value n for the security parameter:

The inverting experiment $\mathsf{Invert}_{\mathcal{A},f}(n)$

1. *Choose input* $x \leftarrow \{0,1\}^n$. *Compute* $y := f(x)$.
2. $\mathcal{A}$ *is given* 1^n *and* y *as input, and outputs* x'.
3. *The output of the experiment is defined to be 1 if* $f(x') = y$, *and 0 otherwise.*

We stress that $\mathcal{A}$ need not find x itself; it suffices for $\mathcal{A}$ to find any value x' for which $f(x') = y = f(x)$. We give the security parameter 1^n to $\mathcal{A}$ in the second step for technical reasons: we want to allow $\mathcal{A}$ to run in time polynomial in the security parameter n, irrespective of the length of y.

We can now define what it means for a function f to be one way.

DEFINITION 6.1 *A function* $f : \{0,1\}^* \to \{0,1\}^*$ *is* one-way *if the following two conditions hold:*

1. (Easy to compute:) *There exists a polynomial-time algorithm* M_f *computing* f*; that is,* $M_f(x) = f(x)$ *for all* x.
2. (Hard to invert:) *For every probabilistic polynomial-time algorithm* $\mathcal{A}$, *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1] \leq \mathsf{negl}(n).$$

Notation: In this chapter we will often make the probability space (more) explicit by subscripting it in the probability. Using this notation, for example, we can very succinctly express the second requirement in the definition above as follows: For every probabilistic polynomial-time algorithm $\mathcal{A}$, there exists a negligible function negl such that

$$\Pr_{x \leftarrow \{0,1\}^n}\left[\mathcal{A}(f(x)) \in f^{-1}(f(x))\right] \leq \mathsf{negl}(n).$$

Successful inversion of one-way functions. A function that is *not* one-way is not necessarily easy to invert all the time (or even "often"). Rather, the converse of the second condition of Definition 6.1 is that there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a non-negligible function ε such that $\mathcal{A}$ inverts $f(x)$ with probability at least $\varepsilon(n)$ (where the probability is taken over random choice of $x \leftarrow \{0,1\}^n$). This means, in turn, that there exists a positive polynomial $q(\cdot)$ such that for *infinitely many values of* n, the algorithm $\mathcal{A}$ inverts f with probability at least $1/q(n)$. Thus, if there exists an $\mathcal{A}$ that inverts f with probability n^{-10} for all even values of n (but always

fails to invert f when n is odd), then f is not one-way. This holds even though $\mathcal{A}$ only succeeds on half the values of n, and even though $\mathcal{A}$ only succeeds with probability n^{-10} (for values of n where it succeeds at all).

Exponential-time inversion. Any one-way function can be inverted given enough time. Specifically, given a value y and the security parameter 1^n, it is always possible to simply try all values $x \in \{0,1\}^n$ until a value x is found such that $f(x) = y$. This algorithm runs in exponential time and always succeeds. Thus, the existence of one-way functions is inherently an assumption about *computational complexity* and *computational hardness*. That is, it considers a problem that can be solved in principle but is assumed to be hard to solve efficiently.

One-way permutations. We will often be interested in one-way functions with additional structural properties. We say a function is *length-preserving* if $|f(x)| = |x|$ for all x. A one-way function that is length-preserving and one-to-one is called a *one-way permutation*. Formally:

DEFINITION 6.2 *Let $f : \{0,1\}^* \to \{0,1\}^*$ be length-preserving, and let f_n be the restriction of f to the domain $\{0,1\}^n$ (i.e., f_n is only defined for $x \in \{0,1\}^n$, in which case $f_n(x) = f(x)$). A one-way function f is called a* one-way permutation *if for every n, the function f_n is a bijection.*

An interesting property of one-way permutations is that any value y uniquely determines its pre-image $x = f^{-1}(y)$. Even though y fully determines x, it is still hard to find x in polynomial time.

Families of one-way functions and permutations. The above definitions of one-way functions and permutations are very convenient in that they consider a single function over an infinite domain and range. However, most candidate one-way functions and permutations that we know of do not fit naturally into this framework. Rather, there is typically an algorithm that generates some parameters I which define some function f_I; the requirement is essentially that f_I should be one-way with all but negligible probability over choice of I. Because each value of I defines a different function, we now refer to *families* of one-way functions (resp., permutations). We give the definition now, and refer the reader to the next section for a concrete example (see also Section 7.4.1).

DEFINITION 6.3 *A tuple $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ of probabilistic polynomial-time algorithms is a* family of functions *if the following hold:*

1. *The* parameter-generation algorithm Gen*, on input 1^n, outputs parameters I with $|I| \geq n$. Each value of I output by Gen defines sets $\mathcal{D}_I$ and $\mathcal{R}_I$ that constitute the domain and range, respectively, of a function f_I defined below.*

2. *The* sampling algorithm Samp*, on input I, outputs a uniformly distributed element of $\mathcal{D}_I$ (except possibly with probability negligible in $|I|$).*
3. *The deterministic* evaluation algorithm f*, on input I and $x \in \mathcal{D}_I$, outputs an element $y \in \mathcal{R}_I$. We write this as $y := f_I(x)$.*

Π *is a* family of permutations *if for each value of I output by* $\mathsf{Gen}(1^n)$*, it holds that $\mathcal{D}_I = \mathcal{R}_I$ and the function $f_I : \mathcal{D}_I \to \mathcal{D}_I$ is a bijection.*

Let Π be a family of functions. What follows is the obvious analogue of the experiment introduced previously.

The inverting experiment $\mathsf{Invert}_{\mathcal{A},\Pi}(n)$**:**

1. $\mathsf{Gen}(1^n)$ *is run to obtain I, and then* $\mathsf{Samp}(I)$ *is run to obtain a random $x \leftarrow \mathcal{D}_I$. Finally, $y := f_I(x)$ is computed.*
2. *$\mathcal{A}$ is given I and y as input, and outputs x'.*
3. *The output of the experiment is defined to be 1 if $f_I(x') = y$, and 0 otherwise.*

A function family is one-way if success in the above experiment occurs with negligible probability.

DEFINITION 6.4 *A function/permutation family $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ is* one-way *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Invert}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

Throughout this chapter we work with one-way functions and permutations as per Definitions 6.1 and 6.2, rather than working with families of one-way functions. This is primarily for convenience, and does not significantly affect any of the results. See also Exercise 6.6.

6.1.2 Candidate One-Way Functions

One-way functions are of interest only if they exist. Since we do not know how to prove that they exist unconditionally (because this would imply a major breakthrough in complexity theory), we conjecture or assume their existence. This conjecture (or assumption) is based on some very natural computational problems that have received much attention, and have yet to yield polynomial-time algorithms. Perhaps the most famous of these problems is that of *integer factorization*, i.e., finding the prime factors of a large integer. This leads us to define the function $f_{\text{mult}}(x, y) = x \cdot y$. (Formally, on input a string of length n, we let x be the first $\lfloor n/2 \rfloor$ bits and y be the last $\lceil n/2 \rceil$ bits; the output is xy.) Note, however, that if we do not place any restriction

on the lengths of x and y, then f_{mult} is easy to invert: with high probability $x \cdot y$ will be *even* in which case the factors $(2, xy/2)$ can be returned as an inverse. There are two ways to modify f_{mult} to address this problem. The more direct way to do this is to simply include the lengths of x and y (when viewed as integers, ignoring leading 0s) as part of the output; i.e., to define

$$f_{\text{mult}}(x, y) = (xy, \|x\|, \|y\|).$$

Alternatively, as described above, we can require that x and y are always both of length exactly $n/2$. A second approach is, in essence, to restrict the domain of f_{mult} to equal-length *primes* x and y. We return to this idea in Chapter 7.

Another candidate one-way function, not relying directly on number theory, is based on the *subset-sum problem* and is defined by

$$f(x_1, \ldots, x_n, J) = (x_1, \ldots, x_n, \sum_{j \in J} x_j),$$

where each x_i is an n-bit string interpreted as an integer, and J is an n-bit string interpreted as a subset of $\{1, \ldots, n\}$. Given an output $(x_1, \ldots, x_n, y)$ of this function, the task of inverting it is exactly that of finding a subset $J' \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in J'} x_j = y$. Students who have studied $\mathcal{NP}$-completeness may be familiar with the fact that this problem is $\mathcal{NP}$-complete. We stress that this does not mean that $\mathcal{P} \neq \mathcal{NP}$ implies the existence of one-way functions: $\mathcal{P} \neq \mathcal{NP}$ would mean that the subset-sum problem cannot be solved *in the worst-case* in polynomial-time, while a one-way function is required to be hard to invert *almost always*. Thus, our belief that the function above is one-way is based on the lack of known algorithms to solve this problem, and not merely on the fact that the general problem is $\mathcal{NP}$-complete.

We conclude by showing a family of *permutations* that is believed to be one-way. Let Gen be a probabilistic polynomial-time algorithm that, on input 1^n, outputs an n-bit prime p along with a special element $g \in \{2, \ldots, p-1\}$. (We will see in Chapter 7 that such algorithms exist. We leave undefined for now what we mean when we say that g is "special".) Let Samp be an algorithm that given p and g outputs a random integer x in the range $\{1, \ldots, p-1\}$. Finally, define

$$f_{p,g}(x) = g^x \bmod p$$

for $x \in \{1, \ldots, p-1\}$. The fact that $f_{p,g}$ can be computed efficiently follows from the results in Appendix B.2.3. It can be shown that this function is one-to-one, and thus a permutation. The presumed difficulty of inverting this function is based on the conjectured hardness of the *discrete logarithm problem*; we will have much more to say about this in Chapter 7.

6.1.3 Hard-Core Predicates

By definition, a one-way function is hard to invert. Stated differently, given a value $y = f(x)$, the value of x cannot be determined *in its entirety* by any

polynomial-time algorithm. This may give the impression that nothing about x can be determined in polynomial time from $f(x)$. However, this is *not* the case. Indeed, it is possible that $f(x)$ "leaks" a lot of information about x, and yet f is still hard to invert. For a trivial example, let f be a one-way function and define $g(x_1, x_2) = (x_1, f(x_2))$, where $|x_1| = |x_2|$. It is easy to show that g is also a one-way function (this is left as an exercise), even though it reveals half its input.

For our applications, we will need to find *some* information about x that is hidden by $f(x)$. This motivates the notion of a *hard-core predicate*. Loosely speaking, a hard-core predicate $\mathsf{hc} : \{0,1\}^* \to \{0,1\}$ of a function f has the following property: given $f(x)$, it is infeasible for any polynomial-time algorithm to correctly determine $\mathsf{hc}(x)$ with probability significantly better than 1/2. (It is always possible to compute $\mathsf{hc}(x)$ correctly with probability exactly 1/2 by random guessing.)

DEFINITION 6.5 *A function* $\mathsf{hc} : \{0,1\}^* \to \{0,1\}$ *is a* hard-core predicate of a function f *if* **(1)** hc *can be computed in polynomial time, and* **(2)** *for every probabilistic polynomial-time algorithm* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr_{x \leftarrow \{0,1\}^n}[\mathcal{A}(f(x)) = \mathsf{hc}(x)] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over the uniform choice of x *in* $\{0,1\}^n$ *and the random coin tosses of* $\mathcal{A}$.

We stress that $\mathsf{hc}(x)$ is efficiently computable given x (since the function hc can be computed in polynomial time); the definition requires that $\mathsf{hc}(x)$ is hard to compute given $f(x)$. The above definition does not require f to be one-way, though we will only be interested in hard-core predicates when that is the case.

Simple ideas don't work. Consider for a moment the candidate hard-core predicate defined as $\mathsf{hc}(x) = \bigoplus_{i=1}^{n} x_i$ where $x_1, \ldots, x_n$ denote the bits of x. The intuition behind why this function "should" be a hard-core predicate is that if f cannot be inverted, then $f(x)$ must hide at least one of the bits x_i of its pre-image. Then, the exclusive-or of all of the bits of x must be hard to compute (since x_i alone is already hard to compute). Despite its appeal, this argument is incorrect. Specifically, given a one-way function f, define the function $g(x) = (f(x), \bigoplus_{i=1}^{n} x_i)$. It is not hard to show that g is one-way. However, it is clear that $g(x)$ does not hide the value of $\mathsf{hc}(x) = \bigoplus_{i=1}^{n} x_i$ because this is part of its output; therefore, $\mathsf{hc}(x)$ is not always a hard-core predicate. (By extending this argument, it can be shown that for any given predicate hc, there exists a one-way function f for which hc is *not* a hard-core predicate of f.)

Trivial hard-core predicates. Some functions have "trivial" hard-core predicates. For example, let f be the function that simply drops the last bit of its input (i.e., $f(x_1 \cdots x_n) = x_1 \cdots x_{n-1}$)). It is immediate that it is hard to predict x_n given $f(x) = x_1 \cdots x_{n-1}$ since x_n is independent of the output. However, f is not one-way. When we use hard-core predicates to construct pseudorandom generators, it will become clear why trivial hard-core predicates of this sort are of no use for cryptography.

In contrast, a one-to-one function f that has a hard-core predicate must be one-way (see Exercise 6.10). Intuitively, this is the case because when a function is one-to-one, the value $f(x)$ fully determines x in an information-theoretic sense. Thus, inability to compute $\mathsf{hc}(x)$ from $f(x)$ must be due to some computational limitation in determining x from $f(x)$.

6.2 Overview: From One-Way Functions to Pseudorandom Permutations

The goal of this chapter is to show how to construct pseudorandom generators, functions, and permutations based on any one-way permutation. In this section, we give an overview of these constructions. Details are given in the sections that follow.

A hard-core predicate for any one-way function. The first step is to show that a hard-core predicate exists for any one-way function. Actually, it remains open whether such a statement is true; we will show something slightly weaker that suffices for our purposes. Namely, we will show that given any one-way function f we can construct a *different* one-way function g along with a hard-core predicate for g. That is:

THEOREM 6.6 *Let f be a one-way function. Then there exists (constructively) a one-way function g along with a hard-core predicate* gl *for g. Furthermore, if f is a permutation then so is g.*

(The hard-core predicate is denoted gl after Goldreich and Levin who proved Theorem 6.6.) Functions g and gl are constructed as follows: set $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, for $|x| = |r|$, and define

$$\mathsf{gl}(x,r) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{n} x_i \cdot r_i,$$

where $x = x_1 \cdots x_n$ (and similarly for r). Notice that the function $\mathsf{gl}(x,\cdot)$ outputs the exclusive-or of a *random subset* of the bits of x. This is due to the fact that r can be viewed as selecting a random subset of $\{1,\ldots,n\}$ (i.e.,

when $r_i = 1$ the bit x_i is included in the XOR, and otherwise it is not), and r is uniformly distributed. Thus, Theorem 6.6 essentially states that if f is an arbitrary one-way function, then $f(x)$ hides the exclusive-or of a *random subset* of the bits of x.

Pseudorandom generators from one-way permutations. The next step is to show how the hard-core predicate of a one-way *permutation* can be used to construct a pseudorandom generator. (It is known that one-way *functions* suffice for constructing pseudorandom generators, but the proof is extremely complicated and well beyond the scope of this book.) Specifically, we show the following:

THEOREM 6.7 *Let f be a one-way permutation and let hc be a hard-core predicate of f. Then, $G(s) \stackrel{\text{def}}{=} (f(s), \mathsf{hc}(s))$ constitutes a pseudorandom generator with expansion factor $\ell(n) = n + 1$.*

As intuition for why G as defined in the theorem constitutes a pseudorandom generator, note first that the initial n bits of the output of $G(s)$ (i.e., the bits of $f(s)$) are *truly* random when s is chosen uniformly at random, by virtue of the fact that f is a permutation. Next, the fact that hc is a hard-core predicate means that $\mathsf{hc}(s)$ "looks random" — i.e., is *pseudorandom* — even given $f(s)$ (assuming again that s is chosen at random). Putting these observations together we see that the entire output of G is pseudorandom.

Pseudorandom generators with arbitrary expansion. The existence of a pseudorandom generator that stretches its seed by even a single bit (as we have just seen) is already highly non-trivial. But for applications (e.g., for efficient encryption of large messages as in Section 3.4), we need a pseudorandom generator with much larger expansion factor. Fortunately, we can obtain an expansion factor that is essentially as long as we like:

THEOREM 6.8 *Assume that there exists a pseudorandom generator with expansion factor $\ell(n) = n + 1$. Then for any polynomial $p(\cdot)$, there exists a pseudorandom generator with expansion factor $\ell(n) = p(n)$.*

We conclude that pseudorandom generators with (essentially) arbitrary expansion factor can be constructed from any one-way permutation.

Pseudorandom functions and permutations from pseudorandom generators. Pseudorandom generators suffice for obtaining private-key encryption schemes with indistinguishable encryptions in the presence of an eavesdropper. For achieving CPA-secure private-key encryption (not to mention message authentication codes), however, we relied on pseudorandom functions. The following result shows that the latter can be constructed from the former:

THEOREM 6.9 *Assume that there exists a pseudorandom generator with expansion factor $\ell(n) = 2n$. Then there exist pseudorandom functions.*

In fact, we can do even more:

THEOREM 6.10 *Assume that there exist pseudorandom functions. Then there exist strong pseudorandom permutations.*

Combining all the above theorems, as well as the results of Chapters 3 and 4, we have the following corollaries:

COROLLARY 6.11 *Assuming the existence of one-way permutations, there exist pseudorandom generators with any polynomial expansion factor, pseudorandom functions, and strong pseudorandom permutations.*

COROLLARY 6.12 *Assuming the existence of one-way permutations, there exist CCA-secure private-key encryption schemes, and message authentication codes that are existentially unforgeable under an adaptive chosen message attack.*

As noted earlier, it is actually possible to obtain all these results based solely on the existence of one-way *functions*.

6.3 A Hard-Core Predicate for Any One-Way Function

In this section, we prove Theorem 6.6 by showing the following:

THEOREM 6.13 *Let f be a one-way function and define g by $g(x, r) \stackrel{\text{def}}{=} (f(x), r)$, where $|x| = |r|$. Define $\mathsf{gl}(x, r) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{n} x_i \cdot r_i$, where $x = x_1 \cdots x_n$ and $r = r_1 \cdots r_n$. Then gl is a hard-core predicate of the function g.*

We now proceed to prove Theorem 6.13. Due to the complexity of the proof, we prove three successively stronger results culminating with what is claimed in the theorem.

6.3.1 A Simple Case

We first show that if there exists a polynomial-time adversary $\mathcal{A}$ that *always* correctly computes $\mathsf{gl}(x, r)$ given $g(x, r) = (f(x), r)$, then it is possible

to invert f in polynomial time. Given the assumption that f is a one-way function, it follows that no such adversary $\mathcal{A}$ exists.

PROPOSITION 6.14 *Let f and gl be as in Theorem 6.13. If there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ such that*

$$\Pr_{x,r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]=1$$

for infinitely-many values of n, then there exists a probabilistic polynomial-time algorithm $\mathcal{A}'$ such that

$$\Pr_{x\leftarrow\{0,1\}^n}\big[\mathcal{A}'(f(x))\in f^{-1}(f(x))\big]=1$$

for infinitely-many values of n.

PROOF Let $\mathcal{A}$ be as in the proposition. We construct $\mathcal{A}'$ as follows. On input y with $|y|=n$, adversary $\mathcal{A}'$ computes $x_i:=\mathcal{A}(y,e^i)$ for $i=1,\ldots,n$, where e^i denotes the n-bit string with 1 in the ith position and 0 everywhere else. Then $\mathcal{A}'$ outputs $x=x_1\cdots x_n$. Clearly $\mathcal{A}'$ runs in polynomial time.

To analyze the success of $\mathcal{A}'$ in inverting f, fix an n for which

$$\Pr_{x,r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]=1 \tag{6.1}$$

and consider the execution of $\mathcal{A}'(y)$. Denote $y=f(\hat{x})$. Then, the value x_i computed by $\mathcal{A}'$ satisfies

$$x_i=\mathcal{A}(f(\hat{x}),e^i)=\bigoplus_{j=1}^{n}\hat{x}_j\cdot e^i_j=\hat{x}_i,$$

using the definition of $\mathsf{gl}(x,r)$ for the second equality, and the fact that $e^i_j=0$ for all $j\neq i$ for the third equality. Thus, $x_i=\hat{x}_i$ for all i and so $\mathcal{A}'$ outputs the correct inverse $x=\hat{x}$ with probability 1. ■

By the assumption that f is one-way, it is impossible for any probabilistic polynomial-time algorithm to invert f with non-negligible probability. Thus, we conclude that there is no probabilistic polynomial-time algorithm that always correctly computes $\mathsf{gl}(x,r)$ from $(f(x),r)$ for infinitely-many values of n. This is a rather weak result that is very far from our ultimate goal of showing that $\mathsf{gl}(x,r)$ cannot be determined with probability significantly better than 1/2.

6.3.2 A More Involved Case

We now show that it is hard for any polynomial-time algorithm $\mathcal{A}$ to compute $\mathsf{gl}(x,r)$ with probability significantly better than 3/4. Assuming such

an $\mathcal{A}$ exists, we will once again show that this implies the existence of a polynomial-time $\mathcal{A}'$ that inverts f with non-negligible probability. Notice that the strategy in the proof of Proposition 6.14 fails completely here because it may be that $\mathcal{A}$ *never* succeeds when $r = e^i$ (though it may succeed, say, on all other values of r). Furthermore, in the present case $\mathcal{A}'$ does not know if a particular bit output by $\mathcal{A}$ as a guess for $\mathsf{gl}(x, r)$ is correct or not — the only thing $\mathcal{A}'$ knows is that with probability non-negligibly greater than 3/4, adversary $\mathcal{A}$ is correct. These issues further complicate the proof.

PROPOSITION 6.15 *Let f and gl be as in Theorem 6.13. If there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a polynomial $p(\cdot)$ such that*

$$\Pr_{x,r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r) = \mathsf{gl}(x,r)\Big] \geq \frac{3}{4} + \frac{1}{p(n)}$$

for infinitely-many values of n, then there exists a probabilistic polynomial-time algorithm $\mathcal{A}'$ such that

$$\Pr_{x\leftarrow\{0,1\}^n}\big[\mathcal{A}'(f(x)) \in f^{-1}(f(x))\big] \geq \frac{1}{4\cdot p(n)}$$

for infinitely-many values of n.

PROOF The main observation underlying the proof of this proposition is that for every $r \in \{0,1\}^n$, the values $\mathsf{gl}(x, r \oplus e^i)$ and $\mathsf{gl}(x, r)$ together can be used to derive the ith bit of x. (Recall that e^i denotes the n-bit string with 0s everywhere except the ith position.) This follows from the following calculation:

$$\begin{aligned}&\mathsf{gl}(x,r) \oplus \mathsf{gl}(x, r\oplus e^i)\\ &\quad= \left(\bigoplus_{j=1}^{n} x_j \cdot r_j\right) \oplus \left(\bigoplus_{j=1}^{n} x_j \cdot (r_j \oplus e^i_j)\right) = x_i \cdot r_i \oplus \big(x_i \cdot (r_i \oplus 1)\big) = x_i\,,\end{aligned}$$

where the second equality is due to the fact that for all $j \neq i$, the value $x_j \cdot r_j$ appears in both sums and so is canceled out.

The above illustrates that if $\mathcal{A}$ answers correctly on both $(f(x), r)$ and $(f(x), r \oplus e^i)$, then $\mathcal{A}'$ can correctly compute x_i. Unfortunately, $\mathcal{A}'$ does not know when $\mathcal{A}$ answers correctly and when it does not; it only knows that $\mathcal{A}$ answers correctly with "high" probability. For this reason, $\mathcal{A}'$ will use multiple random values of r, using each one to obtain a guess for x_i, and will then take the majority value as its final guess of x_i. As a preliminary step, we therefore show that for many x's, the probability that $\mathcal{A}$ answers correctly for both $(f(x), r)$ and $(f(x), r \oplus e^i)$, when r is chosen uniformly at random, is sufficiently high. This is proved in the following claims. These claims will

allow us to fix x and then focus solely on the uniform choice of r, which makes the analysis easier.

CLAIM 6.16 *Let n be such that*

$$\Pr_{x,r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]\geq\frac{3}{4}+\frac{1}{p(n)}\,.$$

Then there exists a set $S_n\subseteq\{0,1\}^n$ of size at least $\frac{1}{2p(n)}\cdot 2^n$ such that for every $x\in S_n$ it holds that

$$\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)]\geq\frac{3}{4}+\frac{1}{2p(n)}\,. \tag{6.2}$$

PROOF Set $\varepsilon(n)=1/p(n)$ and for any $x\in\{0,1\}^n$, let

$$s(x)\stackrel{\text{def}}{=}\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)].$$

Let S_n be the set of all x's for which $s(x)\geq 3/4+\varepsilon(n)/2$ (i.e., for which Equation (6.2) holds). If $S_n=\{0,1\}^n$ we are done. Otherwise, we show that $|S_n|\geq\frac{\varepsilon(n)}{2}\cdot 2^n$ using a simple averaging argument. We have:

$$\begin{aligned}
&\Pr_{x,r}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)]\\
&\quad=\Pr_{x,r}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\mid x\in S_n]\cdot\Pr_{x}[x\in S_n]\\
&\qquad+\Pr_{x,r}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\mid x\notin S_n]\cdot\Pr_{x}[x\notin S_n]\\
&\quad\leq\Pr_{x}[x\in S_n]+\Pr_{x,r}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\mid x\notin S_n],
\end{aligned}$$

where subscripted variables (i.e., x and/or r) indicate those being chosen at random from $\{0,1\}^n$, while non-subscripted variables are fixed. Therefore:

$$\begin{aligned}
&\Pr_{x}[x\in S_n]\geq\\
&\qquad\Pr_{x,r}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)]-\Pr_{x,r}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\mid x\notin S_n].
\end{aligned}$$

By definition of S_n, for every $x\notin S_n$, $\Pr[A(f(x),r)=\mathsf{gl}(x,r)]<3/4+\varepsilon(n)/2$. That is, $\Pr_{x,r}[A(f(x),r)=\mathsf{gl}(x,r)\mid x\notin S_n]<3/4+\varepsilon(n)/2$, and so

$$\Pr_{x}[x\in S_n]\geq\frac{3}{4}+\varepsilon(n)-\left(\frac{3}{4}+\frac{\varepsilon(n)}{2}\right)=\frac{\varepsilon(n)}{2}\,.$$

This implies that S_n must be of size at least $\frac{\varepsilon(n)}{2}\cdot 2^n$ (because x is uniformly distributed in $\{0,1\}^n$), completing the proof of the claim. ■

The following, which is the result we need, now follows as an easy corollary.

CLAIM 6.17 *Let n be such that*

$$\Pr_{x,r\leftarrow\{0,1\}^n}\left[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\right]\geq\frac{3}{4}+\frac{1}{p(n)}\,.$$

Then there exists a set $S_n\subseteq\{0,1\}^n$ of size at least $\frac{1}{2p(n)}\cdot 2^n$ such that for every $x\in S_n$ and every i it holds that

$$\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\wedge\mathcal{A}(f(x),r\oplus e^i)=\mathsf{gl}(x,r\oplus e^i)]\geq\frac{1}{2}+\frac{1}{p(n)}\,.$$

PROOF Let $\varepsilon(n)=1/p(n)$, and take S_n to be the set guaranteed by the previous claim. We know that for any $x\in S_n$ we have

$$\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r)\neq\mathsf{gl}(x,r)]\leq\frac{1}{4}-\varepsilon(n)/2\,.$$

Fix any $i\in\{1,\ldots,n\}$. If r is uniformly distributed then so is $r\oplus e^i$; this means that

$$\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r\oplus e^i)\neq\mathsf{gl}(x,r\oplus e^i)]\leq\frac{1}{4}-\varepsilon(n)/2\,.$$

We are interested in lower-bounding the probability that $\mathcal{A}$ outputs the correct answer for *both* $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x,r\oplus e^i)$; equivalently, we want to upper-bound the probability that $\mathcal{A}$ fails to output the correct answer in *either* of these cases. Note that r and $r\oplus e^i$ are not independent, and so we cannot just multiply the probabilities of failure. However, we can apply the union bound (see Proposition A.7 in Appendix A) and just sum the probabilities of failure. That is, the probability that $\mathcal{A}$ is *incorrect* on either $\mathsf{gl}(x,r)$ or $\mathsf{gl}(x,r\oplus e^i)$ is at most

$$\left(\frac{1}{4}-\frac{\varepsilon(n)}{2}\right)+\left(\frac{1}{4}-\frac{\varepsilon(n)}{2}\right)=\frac{1}{2}-\varepsilon(n),$$

and so $\mathcal{A}$ is correct on *both* $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x,r\oplus e^i)$ with probability *at least* $1/2+\varepsilon(n)$. This proves the claim. ■

For the rest of the proof we set $\varepsilon(n)=1/p(n)$ and consider only those values of n for which $\mathcal{A}$ succeeds with probability at least $3/4+\varepsilon(n)$. The claim above states that for an $\varepsilon(n)/2$ fraction of inputs x, the adversary $\mathcal{A}$ answers correctly on both $(f(x),r)$ and $(f(x),r\oplus e^i)$ with probability at least $1/2+\varepsilon(n)$ over random choice of r, and from now on we will focus only on such values of x. We construct a probabilistic polynomial-time algorithm $\mathcal{A}'$

that inverts $f(x)$ with probability at least $1/2$ when $x \in S_n$. This suffices to prove the proposition since then

$$\begin{aligned}
&\Pr_x[\mathcal{A}'(f(x)) \in f^{-1}(f(x))] \\
&\quad \geq \Pr_x[\mathcal{A}'(f(x)) \in f^{-1}(f(x)) \mid x \in S_n] \cdot \Pr_x[x \in S_n] \\
&\quad \geq \frac{1}{2} \cdot \frac{1}{2p(n)} \;=\; \frac{1}{4p(n)}\,.
\end{aligned}$$

Algorithm $\mathcal{A}'$, given as input an element y, works as follows:

1. For $i = 1, \ldots, n$ do:
   (a) Choose a random $r \leftarrow \{0,1\}^n$ and compute a "guess" that the value $\bar{x}_i := \mathcal{A}(y, r) \oplus \mathcal{A}(y, r \oplus e^i)$ is the ith bit of the pre-image of y.
   (b) Repeat the above sufficiently-many times (see further below), and let x_i be the majority of the guesses.
2. Output $x = x_1 \cdots x_n$.

We sketch an analysis of the probability that $\mathcal{A}'$ correctly inverts its given input y. (We allow ourselves to be a bit laconic, since a full proof for the most difficult case is given in the following section.) Fix n to be (one of the infinitely many values) such that $\Pr_{x,r \leftarrow \{0,1\}^n}[\mathcal{A}(f(x), r) = \mathsf{gl}(x, r)] \geq \frac{3}{4} + \frac{1}{p(n)}$, and assume that $\mathcal{A}'$'s input $y = f(\hat{x})$ is such that $\hat{x} \in S_n$ (recall that the latter occurs with probability at least $\varepsilon(n)/2$). Fix some i. The previous claim implies that the guess $\bar{x}_i$ is equal to $\mathsf{gl}(\hat{x}, e^i)$ with probability at least $\frac{1}{2} + \varepsilon(n)$. By repeating sufficiently-many times and letting x_i be the majority, $\mathcal{A}'$ can ensure that x_i is equal to $\mathsf{gl}(\hat{x}, e^i)$ with probability at least $1 - \frac{1}{2n}$. We need to ensure that this can be done by taking the majority of only *polynomially-many* guesses; since $\varepsilon(n) = 1/p(n)$ for some polynomial p, this is indeed the case as can be shown using a *Chernoff bound* (a standard bound from probability theory), along with the fact that an independent value of r is chosen in each iteration. We leave a full proof using the Chernoff bound as an exercise.

Summarizing where things stand, we have that for each i the value x_i computed by $\mathcal{A}'$ is incorrect with probability at most $\frac{1}{2n}$. A union bound thus shows that $\mathcal{A}'$ is incorrect for *some* i with probability at most $n \cdot \frac{1}{2n} = \frac{1}{2}$. That is, $\mathcal{A}'$ is correct for all i — and thus correctly inverts y — with probability at least $\frac{1}{2}$. This completes the proof of the proposition. ∎

A corollary of Proposition 6.15 is that if f is a one-way function, then the probability of correctly guessing $\mathsf{gl}(x, r)$ when given $(f(x), r)$ is at most negligibly greater than $3/4$. Thus, the bit $\mathsf{gl}(x, r)$ has considerable uncertainty (when considering polynomial-time observers).

6.3.3 The Full Proof

This section is more advanced than the rest of the book, and relies on more involved concepts from probability theory. We include the full proof for completeness, and for more advanced students and courses.

Preliminaries – Probabilistic Sampling

We use some standard results from probability theory that are reviewed quickly here. A 0/1-random variable X_i is one that takes a value in $\{0,1\}$. The 0/1-random variables $X_1, \ldots, X_m$ are *pairwise independent* if for every $i \neq j$ and every $b_i, b_j \in \{0,1\}$ it holds that

$$\Pr[X_i = b_i \ \wedge \ X_j = b_j] = \Pr[X_i = b_i] \cdot \Pr[X_j = b_j].$$

We rely on the following proposition:

PROPOSITION 6.18 *Let $\{X_i\}$ be pairwise-independent, 0/1-random variables with the following property: there exist values $b \in \{0,1\}$ and $\varepsilon > 0$ such that for all $1 \leq i \leq n$,*

$$\Pr[X_i = b] = \frac{1}{2} + \varepsilon.$$

Consider the process in which m values $X_1, \ldots, X_m$ are recorded and X is set to the value that occurs a majority of the time. Then

$$\Pr[X \neq b] \leq \frac{1}{4 \cdot \varepsilon^2 \cdot m}.$$

This proposition is standard. The reader willing to accept the above on faith can proceed directly to the following section; for completeness, we provide a self-contained proof of the proposition here.

Let $\mathsf{Exp}[X]$ denote the expectation of a random variable X. We have:

Markov's Inequality: *Let X be a non-negative random variable and $v > 0$. Then:*

$$\Pr[X \geq v] \leq \mathsf{Exp}[X]/v.$$

PROOF We have

$$\begin{aligned}
\mathsf{Exp}[X] &= \sum_{x \geq 0} \Pr[X = x] \cdot x \\
&\geq \sum_{0 \leq x < v} \Pr[X = x] \cdot 0 + \sum_{x \geq v} \Pr[X = x] \cdot v \\
&= \Pr[X \geq v] \cdot v.
\end{aligned}$$

■

Markov's inequality is useful when very little information about X is known. When an upper-bound on the variance of X is known, however, better bounds exist. We will use the following basic facts from probability: $\mathsf{Var}[X] \stackrel{\text{def}}{=} \mathsf{Exp}[(X-\mathsf{Exp}[X])^2]$, $\mathsf{Var}[X] = \mathsf{Exp}[X^2]-\mathsf{Exp}[X]^2$, and $\mathsf{Var}[aX+b] = a^2\mathsf{Var}[X]$.

Chebyshev's Inequality: *Let X be a random variable and $\delta > 0$. Then:*

$$\Pr[|X - \mathsf{Exp}[X]| \geq \delta] \leq \frac{\mathsf{Var}[X]}{\delta^2}.$$

PROOF Define the non-negative random variable $Y \stackrel{\text{def}}{=} (X-\mathsf{Exp}[X])^2$ and then apply Markov's inequality. That is,

$$\begin{aligned}\Pr[|X - \mathsf{Exp}[X]| \geq \delta] &\leq \Pr[(X - \mathsf{Exp}[X])^2 \geq \delta^2] \\ &\leq \frac{\mathsf{Exp}[(X - \mathsf{Exp}[X])^2]}{\delta^2} \\ &= \frac{\mathsf{Var}[X]}{\delta^2}.\end{aligned}$$

■

If $X_1, \ldots, X_m$ are pairwise-independent then $\mathsf{Var}[\sum_{i=1}^m X_i] = \sum_{i=1}^m \mathsf{Var}[X_i]$ (this is due to the fact that $\mathsf{Exp}[X_i \cdot X_j] = \mathsf{Exp}[X_i] \cdot \mathsf{Exp}[X_j]$ when $i \neq j$, using pairwise independence). An important corollary of Chebyshev's inequality follows.

COROLLARY 6.19 *Let $X_1, \ldots, X_m$ be pairwise-independent random variables with the same expectation μ and the same variance σ^2. For every $\varepsilon > 0$,*

$$\Pr\left[\left|\frac{\sum_{i=1}^m X_i}{m} - \mu\right| \geq \varepsilon\right] \leq \frac{\sigma^2}{\varepsilon^2 m}.$$

PROOF By linearity of expectations, $\mathsf{Exp}[\sum_{i=1}^m X_i/m] = \mu$. Applying Chebyshev's inequality to the random variable $\sum_{i=1}^m X_i/m$, we have

$$\Pr\left[\left|\frac{\sum_{i=1}^m X_i}{m} - \mu\right| \geq \varepsilon\right] \leq \frac{\mathsf{Var}\left[\frac{1}{m} \cdot \sum_{i=1}^m X_i\right]}{\varepsilon^2}.$$

Using pairwise independence, it follows that

$$\mathsf{Var}\left[\frac{1}{m} \cdot \sum_{i=1}^m X_i\right] = \frac{1}{m^2} \sum_{i=1}^m \mathsf{Var}[X_i] = \frac{1}{m^2} \sum_{i=1}^m \sigma^2 = \frac{\sigma^2}{m}.$$

The inequality is obtained by combining the above two equations. ■

We now prove Proposition 6.18. Take $b = 1$ in the proposition (by symmetry, this choice is irrelevant); this means $\mathsf{Exp}[X_i] = \frac{1}{2} + \varepsilon$. Let X denote the majority value of the $\{X_i\}$ as in the proposition, and note that $X \neq 1$ only if $\sum_{i=1}^{m} X_i \leq m/2$. So

$$\begin{aligned}
\Pr[X \neq 1] &\leq \Pr\left[\sum_{i=1}^{m} X_i \leq m/2\right] \\
&= \Pr\left[\frac{\sum_{i=1}^{m} X_i}{m} - \frac{1}{2} \leq 0\right] \\
&= \Pr\left[\frac{\sum_{i=1}^{m} X_i}{m} - \left(\frac{1}{2} + \varepsilon\right) \leq -\varepsilon\right] \\
&\leq \Pr\left[\left|\frac{\sum_{i=1}^{m} X_i}{m} - \left(\frac{1}{2} + \varepsilon\right)\right| \geq \varepsilon\right].
\end{aligned}$$

For a 0/1-random variable X_i, we have $\sigma^2 = \mathsf{Var}[X_i] \leq 1/4$ (this is because in such a case $\mathsf{Exp}[X_i] = \mathsf{Exp}[X_i^2]$ and so $\mathsf{Var}[X_i] = \mathsf{Exp}[X_i](1 - \mathsf{Exp}[X_i])$ which is maximized when $\mathsf{Exp}[X_i] = \frac{1}{2}$). Applying the previous corollary, we conclude that

$$\Pr[X \neq 1] \leq \frac{1}{4\varepsilon^2 m},$$

as claimed.

Proof of Theorem 6.13

We assume familiarity with the simplified proofs in the previous sections, and rely on the ideas developed there. We prove the following proposition, which implies Theorem 6.13:

PROPOSITION 6.20 *Let f and gl be as in Theorem 6.13. If there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a polynomial $p(\cdot)$ such that*

$$\Pr_{x,r \leftarrow \{0,1\}^n}\left[\mathcal{A}(f(x), r) = \mathsf{gl}(x, r)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

for infinitely-many values of n, then there exists a probabilistic polynomial-time adversary $\mathcal{A}'$ and a polynomial $p'(\cdot)$ such that

$$\Pr_{x \leftarrow \{0,1\}^n}\left[\mathcal{A}'(f(x)) \in f^{-1}(f(x))\right] \geq \frac{1}{p'(n)}$$

for infinitely-many values of n.

PROOF As in the proof of Proposition 6.15, we set $\varepsilon(n) = 1/p(n)$ and consider only those values of n for which $\mathcal{A}$ succeeds with probability $1/2 + \varepsilon(n)$. The following is analogous to Claim 6.16 and is proved in the same way.

CLAIM 6.21 *Let n be such that*

$$\Pr_{x,r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]\geq\frac{1}{2}+\varepsilon(n).$$

Then there exists a set $S_n\subseteq\{0,1\}^n$ of size at least $\frac{\varepsilon(n)}{2}\cdot 2^n$ such that for every $x\in S_n$ it holds that

$$s(x)\stackrel{\text{def}}{=}\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)]\geq\frac{1}{2}+\frac{\varepsilon(n)}{2}. \tag{6.3}$$

If we try to proceed exactly as in the proof of Proposition 6.15, we will run into trouble because an analogue of Claim 6.17 will *not* hold here. Specifically, the best we can claim here is that when $x\in S_n$ it holds that

$$\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\wedge\mathcal{A}(f(x),r\oplus e^i)=\mathsf{gl}(x,r\oplus e^i)]\geq\frac{1}{p(n)}$$

for any i. This means that if we try to construct an algorithm $\mathcal{A}'$ that guesses x_i by computing $\mathcal{A}(f(x),r)\oplus\mathcal{A}(f(x),r\oplus e^i)$, then all we can claim is that this guess will be correct with probability at least $1/p(n)$, which is not even any better than taking a random guess! (Moreover, we cannot claim that flipping the result gives a good guess with high probability, either.)

Instead, we design $\mathcal{A}'$ so that it computes $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x,r\oplus e^i)$ by invoking $\mathcal{A}$ only once. We do this by having $\mathcal{A}'$ run $\mathcal{A}(x,r\oplus e^i)$, and having $\mathcal{A}'$ simply "guess" the value $\mathsf{gl}(x,r)$ itself. The naive way to do this would be to choose the r's independently, as before, and to make an independent guess of $\mathsf{gl}(x,r)$ for each value of r. But then the probability that all guesses are correct — which, as we will see, is needed if $\mathcal{A}'$ is to output the correct answer — would be negligible because polynomially-many different r's are used.

The crucial observation of the present proof is that $\mathcal{A}'$ can generate the r's in a *pairwise-independent* manner, and make its guesses in a particular way so that with non-negligible probability all of its guesses are correct. Specifically, in order to generate m different values of r, $\mathcal{A}'$ selects $\ell=\lceil\log(m+1)\rceil$ independent and uniformly-distributed strings $s^1,\ldots,s^\ell\in\{0,1\}^n$. Then, for every non-empty subset $I\subseteq\{1,\ldots,\ell\}$, algorithm $\mathcal{A}'$ sets $r^I:=\oplus_{i\in I}s^i$. Since there are $2^\ell-1$ non-empty subsets, this defines $2^{\lceil\log(m+1)\rceil}-1\geq m$ different strings. Each such string is uniformly distributed when considered in isolation. Moreover, these strings are all pairwise independent. To see this, notice that for every two subsets $I\neq J$ there is an index $j\in I\cup J$ such that $j\notin I\cap J$. Without loss of generality, assume $j\in J$. Then, even conditioned on some known value of r^I, nothing about the value of s^j is revealed and so s^j is still uniformly distributed. Furthermore, since s^j is included in the XOR that defines r^J, we have that r^J is uniformly distributed even conditioned on some known value of r^I.

We now have the following two important observations:

1. Given the correct values of $\mathsf{gl}(x, s^1), \ldots, \mathsf{gl}(x, s^\ell)$, it is possible to correctly compute $\mathsf{gl}(x, r^I)$ for every non-empty subset $I \subseteq \{1, \ldots, \ell\}$. This is because
$$\mathsf{gl}(x, r^I) = \mathsf{gl}(x, \oplus_{i \in I}\, s^i) = \oplus_{i \in I}\, \mathsf{gl}(x, s^i).$$
2. The values $\mathsf{gl}(x, s^1), \ldots, \mathsf{gl}(x, s^\ell)$ can all be correctly guessed with probability $1/2^\ell$. This holds because each bit $\mathsf{gl}(x, s^i)$ is guessed correctly with probability $1/2$ and there are ℓ bits. If m is polynomial in the security parameter n, it follows that 2^ℓ is also polynomial in n. Thus, with *non-negligible probability* it is possible to correctly guess all the values $\mathsf{gl}(x, s^1), \ldots, \mathsf{gl}(x, s^\ell)$.

Combining the above, we see that this yields a way of obtaining $m = \mathsf{poly}(n)$ pairwise-independent strings $\{r^I\}$ along with *correct* values for $\{\mathsf{gl}(x, r^I)\}$, for all I, with non-negligible probability. These values can then be used to compute x_i in the same way as in the proof of Proposition 6.15. Details follow.

The inversion algorithm $\mathcal{A}'$. We now provide a full description of an algorithm $\mathcal{A}'$ that receives input y and tries to compute an inverse of y. The algorithm proceeds as follows:

1. Set $n := |y|$ and $\ell := \lceil \log(2n/\varepsilon(n)^2 + 1) \rceil$.
2. Choose $s^1, \ldots, s^\ell \leftarrow \{0,1\}^n$ and $\sigma^1 \ldots, \sigma^\ell \leftarrow \{0,1\}$ uniformly at random.
3. For every non-empty subset $I \subseteq \{1, \ldots, \ell\}$, set $r^I := \oplus_{i \in I}\, s^i$ and compute $\rho^I := \oplus_{i \in I}\, \sigma^i$.
4. For $i = 1, \ldots, n$:
   (a) For every non-empty subset $I \subseteq \{1, \ldots, \ell\}$, set
   $$x_i^I := \rho^I \oplus \mathcal{A}(y, r^I \oplus e^i).$$
   (b) Set $x_i := \mathsf{majority}_I\{x_i^I\}$ (i.e., take the bit that appeared a majority of the times in the previous step).
5. Output $x = x_1 \cdots x_n$.

Analyzing the success probability of $\mathcal{A}'$. It remains to compute the probability that $\mathcal{A}'$ successfully outputs $x \in f^{-1}(y)$. Similarly to the proof of Proposition 6.15, we focus only on the case when $y = f(\hat{x})$ for $\hat{x} \in S_n$. Each σ^i can be viewed as a "guess" for the value of $\mathsf{gl}(\hat{x}, s^i)$. As noted earlier, with non-negligible probability all these guesses will be correct; we show that when this occurs then $\mathcal{A}'$ outputs $x = \hat{x}$ with probability at least $1/2$. This will complete the proof.

Assuming $\sigma^i = \mathsf{gl}(\hat{x}, s^i)$ for all i, each ρ^I is equal to the correct value of $\mathsf{gl}(\hat{x}, r^I)$. Fix an index $i \in \{1, \ldots, n\}$ and consider the probability that $\mathcal{A}'$ obtains the correct value of x_i. Because $\mathcal{A}(y, r^I \oplus e^i) = \mathsf{gl}(\hat{x}, r^I \oplus e^i)$ with probability at least $\frac{1}{2} + \varepsilon(n)/2$ (this follows from the facts that $\hat{x} \in S_n$ and that $r^I \oplus e^i$, considered in isolation, is a uniformly-distributed string), we know that

$$\Pr[x_i^I = \hat{x}_i] = \Pr\left[\mathcal{A}(f(\hat{x}), r^I \oplus e^i) = \mathsf{gl}(\hat{x}, r^I \oplus e^i)\right] = \frac{1}{2} + \frac{\varepsilon(n)}{2}$$

for all I. Moreover, the $\{x_i^I\}_{I \subseteq \{1,\ldots,\ell\}}$ are pairwise independent because the $\{r^I\}_{I \subseteq \{1,\ldots,\ell\}}$ (and hence the $\{r^I \oplus e^i\}_{I \subseteq \{1,\ldots,\ell\}}$) are pairwise independent. Since x_i is defined as the value that occurs a majority of the time among $\{\hat{x}_i^I\}$, we are now in a position to apply Proposition 6.18. Setting $m = 2^\ell - 1$, we have that

$$\begin{aligned}\Pr[x_i \neq \hat{x}_i] &\leq \frac{1}{4 \cdot (\varepsilon(n)/2)^2 \cdot (2^\ell - 1)} \\ &\leq \frac{1}{4 \cdot (\varepsilon(n)/2)^2 \cdot (2n/\varepsilon(n)^2)} \\ &= \frac{1}{2n}.\end{aligned}$$

The above holds for all i, so by applying a union bound we see that the probability that $x_i \neq \hat{x}_i$ for *some* i is at most $1/2$. That is, $x_i = \hat{x}_i$ for *all* i (and hence $x = \hat{x}$) with probability at least $1/2$.

Putting everything together: with probability at least $\varepsilon(n)/2$ it holds that $y = f(\hat{x})$ with $\hat{x} \in S_n$. Independently of this event, the probability that all of the guesses σ_i are correct is at least

$$\frac{1}{2^\ell} \geq \frac{1}{2 \cdot (2n/\varepsilon(n)^2 + 1)} > \frac{\varepsilon(n)^2}{5n}$$

(the last inequality holds for n large enough). Conditioned on both of the above, $\mathcal{A}'$ outputs an inverse of y with probability at least $1/2$. The overall probability with which $\mathcal{A}'$ inverts its input y is therefore at least $\varepsilon(n)^3/20n = 1/20 \cdot n \cdot p(n)^3$ for infinitely-many values of n. Since $\mathcal{A}'$ runs in polynomial-time, this contradicts the one-wayness of f. ■

6.4 Constructing Pseudorandom Generators

We first show how to construct pseudorandom generators that stretch their input by a single bit, under the assumption that one-way *permutations* exist. We then show how to extend this to obtain any polynomial expansion factor.

6.4.1 Pseudorandom Generators with Minimal Expansion

Let f be a one-way permutation and let hc be a hard-core predicate of f (such a predicate exists by Theorem 6.13). The starting point for the construction is the fact that given $f(s)$ for a random s, it is hard to guess the value of $\mathsf{hc}(s)$ with probability that is non-negligibly higher than 1/2. Thus, intuitively, $\mathsf{hc}(s)$ is a pseudorandom bit. Furthermore, since f is a permutation, $f(s)$ is uniformly distributed (applying a permutation to a uniformly distributed value yields a uniformly distributed value). We therefore conclude that the string $(f(s), \mathsf{hc}(s))$ is pseudorandom and so the algorithm $G(s) = (f(s), \mathsf{hc}(s))$ constitutes a pseudorandom generator.

THEOREM 6.22 *Let f be a one-way permutation, and let hc be a hard-core predicate of f. Then, the algorithm $G(s) = (f(s), \mathsf{hc}(s))$ is a pseudorandom generator with $\ell(n) = n + 1$.*

PROOF Let D be a probabilistic polynomial-time distinguisher, and set

$$\begin{aligned}\varepsilon(n) &\stackrel{\text{def}}{=} \Pr_{s\leftarrow\{0,1\}^n}[D(G(s)) = 1] - \Pr_{r\leftarrow\{0,1\}^{n+1}}[D(r) = 1] \\ &= \Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \mathsf{hc}(s)) = 1] - \Pr_{r\leftarrow\{0,1\}^{n+1}}[D(r) = 1].\end{aligned}$$

Observe that

$$\begin{aligned}\Pr_{r\leftarrow\{0,1\}^{n+1}}[D(r) = 1] &= \Pr_{r\leftarrow\{0,1\}^n, r'\leftarrow\{0,1\}}[D(r, r') = 1] \\ &= \Pr_{s\leftarrow\{0,1\}^n, r'\leftarrow\{0,1\}}[D(f(s), r') = 1] \\ &= \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \mathsf{hc}(s)) = 1] \\ &\quad + \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \overline{\mathsf{hc}}(s)) = 1],\end{aligned}$$

using the fact that f is a permutation for the first equality, and the fact that a random bit r' is equal to $\mathsf{hc}(s)$ with probability exactly 1/2 for the second equality. Thus,

$$\varepsilon(n) = \frac{1}{2}\cdot\left(\Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \mathsf{hc}(s)) = 1] - \Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \overline{\mathsf{hc}}(s)) = 1]\right).$$

Consider the following algorithm $\mathcal{A}$ that is given as input a value $y = f(s)$ and tries to predict the value of $\mathsf{hc}(s)$:

1. Choose $r' \leftarrow \{0, 1\}$ uniformly at random.
2. Run $D(y, r')$. If D outputs 1, output r'; otherwise output $1 - r'$.

By definition of $\mathcal{A}$ we have

$$\begin{aligned}
&\Pr_{s\leftarrow\{0,1\}^n}[\mathcal{A}(f(s)) = \mathsf{hc}(s)] \\
&\quad = \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^n}[\mathcal{A}(f(s)) = \mathsf{hc}(s) \mid r' = \mathsf{hc}(s)] \\
&\qquad + \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^n}[\mathcal{A}(f(s)) = \mathsf{hc}(s) \mid r' \neq \mathsf{hc}(s)] \\
&\quad = \frac{1}{2}\cdot \left(\Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \mathsf{hc}(s)) = 1] + \Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \overline{\mathsf{hc}}(s)) = 0] \right) \\
&\quad = \frac{1}{2}\cdot \left(\Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \mathsf{hc}(s)) = 1] + (1 - \Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \overline{\mathsf{hc}}(s)) = 1]) \right) \\
&\quad = \frac{1}{2} + \frac{1}{2}\cdot \left(\Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \mathsf{hc}(s)) = 1] - \Pr_{s\leftarrow\{0,1\}^n}[D(f(s), \overline{\mathsf{hc}}(s)) = 1] \right) \\
&\quad = \frac{1}{2} + \varepsilon(n).
\end{aligned}$$

Clearly $\mathcal{A}$ runs in polynomial time. Since hc is a hard-core predicate for f, it follows that there exists a negligible function negl for which $\varepsilon(n) \leq \mathsf{negl}(n)$.

An analogous argument shows that $-\varepsilon(n) \leq \mathsf{negl}(n)$. Taken together, this means that

$$\left| \Pr_{s\leftarrow\{0,1\}^n}[D(G(s)) = 1] - \Pr_{r\leftarrow\{0,1\}^{n+1}}[D(r) = 1] \right| \leq \mathsf{negl}(n),$$

completing the proof that G is a pseudorandom generator. ■

6.4.2 Increasing the Expansion Factor

We now show that the expansion factor of a pseudorandom generator can be increased by any polynomial amount. This means that the previous construction (with expansion factor $\ell(n) = n + 1$) suffices for constructing a pseudorandom generator with arbitrary polynomial expansion factor.

THEOREM 6.23 *If there exists a pseudorandom generator $\hat{G}$ with expansion factor $\hat{\ell}(n) = n + 1$, then for any polynomial $p(n) > n$, there exists a pseudorandom generator G with expansion factor $\ell(n) = p(n)$.*

PROOF The idea behind the construction of G from $\hat{G}$ is as follows. Given an initial seed s of length n, the generator $\hat{G}$ can be used to obtain $n + 1$ pseudorandom bits. One of the $n + 1$ bits may be output, and the remaining n bits can be used once again as a seed for $\hat{G}$. The reason that these n bits can be used as a seed is because they are pseudorandom, and therefore essentially

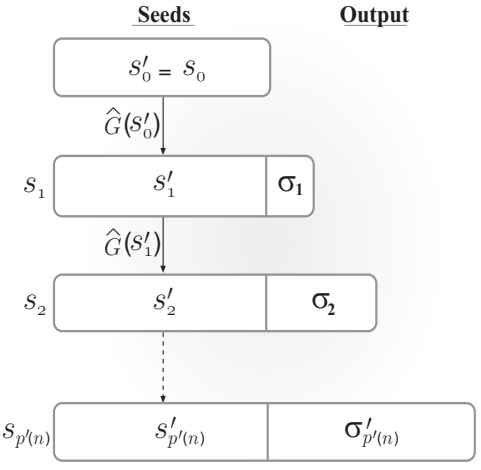

FIGURE 6.1: Increasing the expansion of a pseudorandom generator.

as good as a truly random seed. This procedure can be iteratively applied to output as many bits as desired; see Figure 6.1.

We now formally describe the construction of G. On input $s \in \{0,1\}^n$:

1. Let $p'(n) = p(n) - n$. Note that this is the amount by which G is supposed to increase the length of its input.
2. Set $s_0 := s$. For $i = 1, \ldots, p'(n)$ do:
   (a) Let s'_{i-1} denote the first n bits of s_{i-1}, and let σ_{i-1} denote the remaining $i-1$ bits. (When $i = 1$, σ_0 is the empty string.)
   (b) Set $s_i := (\hat{G}(s'_{i-1}), \sigma_{i-1})$.
3. Output $s_{p'(n)}$.

Before proceeding, note that when $i = 1$, s'_0 is the original seed and in step 2b we have $s_1 = \hat{G}(s'_0)$. Then, when $i = 2$, the string s_1 of length $n+1$ is split into a prefix of length n, denoted s'_1, and a suffix of length 1, denoted σ_1. The string s'_1 is used as the seed to $\hat{G}$ again and the resulting string s_2 is of length $n+2$ (namely, it is $(\hat{G}(s'_1), \sigma_1)$). Observe that in the next iteration, the last two bits of s_2 become σ_2 (where the first bit of σ_2 is the last bit of $\hat{G}(s'_1)$ and the second bit of σ_2 is σ_1). Thus, in each iteration a single extra bit is generated, and this is incorporated into the "σ part". For this reason, the σ_i values grow by one in length in each iteration, as demonstrated in Figure 6.1.

We prove that $G(s)$ is a pseudorandom string of length $p(n)$. We begin by proving this for the simple case of $p(n) = n + 2$.

Define three sequences of distributions $\{H_n^0\}, \{H_n^1\}, \{H_n^2\}$, where each of H_n^1, H_n^2, and H_n^3 is a distribution on strings of length $n+2$. In distribution

H_n^0, the string $s_0 \leftarrow \{0,1\}^n$ is chosen uniformly at random and the output is $G(s_0)$. In distribution H_n^1, the string $s_1 \leftarrow \{0,1\}^{n+1}$ is chosen uniformly at random and then G is run as above but starting from iteration $i = 2$. That is, parse s_1 as (s_1', σ_1) with $|s_1'| = n$, and then output $(\hat{G}(s_1'), \sigma_1)$. In distribution H_n^2, the string $s_2 \leftarrow \{0,1\}^{n+2}$ is chosen uniformly at random and output. We denote by $s_2 \leftarrow H_n^i$ the choice of the $(n+2)$-bit string s_2 according to distribution H_n^i.

We first claim that for any probabilistic polynomial-time distinguisher D there exists a negligible function negl such that

$$\left| \Pr_{s_2 \leftarrow H_n^0}[D(s_2) = 1] - \Pr_{s_2 \leftarrow H_n^1}[D(s_2) = 1] \right| \leq \mathsf{negl}(n). \tag{6.4}$$

To see this, fix some D and consider the polynomial-time distinguisher D' that, on input $w \in \{0,1\}^{n+1}$, sets $s_1 := w$ and then runs G as above but starting from iteration $i = 2$. This yields a string s_2, and then D' outputs $D(s_2)$. The following observations are immediate from the syntactic definitions of H_n^0 and H_n^1:

1. If w is chosen uniformly at random, the distribution on s_2 generated by D' is exactly that of distribution H_n^1. Thus,

$$\Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] = \Pr_{s_2 \leftarrow H_n^1}[D(s_2) = 1].$$

2. If $w = \hat{G}(s)$ for $s \leftarrow \{0,1\}^n$ chosen uniformly at random, the distribution on s_2 generated by D' is exactly that of distribution H_n^0. I.e.,

$$\Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1] = \Pr_{s_2 \leftarrow H_n^0}[D(s_2) = 1].$$

Pseudorandomness of $\hat{G}$ implies that there exists a negligible function negl such that

$$\left| \Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1] - \Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] \right| \leq \mathsf{negl}(n).$$

Equation (6.4) follows.

We next claim that for any probabilistic polynomial-time distinguisher D there exists a negligible function negl such that

$$\left| \Pr_{s_2 \leftarrow H_n^1}[D(s_2) = 1] - \Pr_{s_2 \leftarrow H_n^2}[D(s_2) = 1] \right| \leq \mathsf{negl}(n). \tag{6.5}$$

The proof is very similar. Consider the polynomial-time distinguisher D' that, on input $w \in \{0,1\}^{n+1}$, chooses $\sigma_1 \leftarrow \{0,1\}$ uniformly at random, sets $s_2 := (w, \sigma_1)$, and outputs $D(s_2)$. Notice that if w is chosen uniformly at

random then s_2 is uniformly distributed and so is distributed exactly according to H_n^2. Thus,

$$\Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] = \Pr_{s_2 \leftarrow H_n^2}[D(s_2) = 1].$$

On the other hand, if $w = \hat{G}(s)$ for $s \leftarrow \{0,1\}^n$ chosen uniformly at random then s_2 is distributed exactly according to H_n^1 and so

$$\Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1] = \Pr_{s_2 \leftarrow H_n^1}[D(s_2) = 1].$$

As before, pseudorandomness of $\hat{G}$ implies Equation (6.5).

Fix some probabilistic polynomial-time distinguisher D. We have

$$\begin{aligned} &\left| \Pr_{s \leftarrow \{0,1\}^n}[D(G(s)) = 1] - \Pr_{r \leftarrow \{0,1\}^{n+2}}[D(r) = 1] \right| \qquad (6.6) \\ &= \left| \Pr_{s_2 \leftarrow H_n^0}[D(s_2) = 1] - \Pr_{s_2 \leftarrow H_n^2}[D(s_2) = 1] \right| \\ &\leq \left| \Pr_{s_2 \leftarrow H_n^0}[D(s_2) = 1] - \Pr_{s_2 \leftarrow H_n^1}[D(s_2) = 1] \right| \\ &\quad + \left| \Pr_{s_2 \leftarrow H_n^1}[D(s_2) = 1] - \Pr_{s_2 \leftarrow H_n^2}[D(s_2) = 1] \right|. \end{aligned}$$

Using Equations (6.4) and (6.5), we conclude that Equation (6.6) is negligible.

The full proof. We now give a proof for arbitrary $p(n)$. The main difference here is a technical one: in the case of $p(n) = n + 2$ we could define and explicitly work with three sequences of distributions $\{H_n^0\}$, $\{H_n^1\}$, and $\{H_n^2\}$. Here, in contrast, we will (in some sense) need to deal with infinitely-many sequences of distributions. Instead of dealing with these explicitly, we deal with them *implicitly* using a common technique known as a *hybrid argument*. (Actually, even the case of $p(n) = 2$ utilized a simple hybrid argument.)

Let $p'(n) = p(n) - n$. For any n and $0 \leq j \leq p'(n)$, let H_n^j be the distribution on strings of length $p(n)$ defined as follows: choose $s_j \leftarrow \{0,1\}^{n+j}$ uniformly at random and then run G starting from iteration $i = j + 1$ and output $s_{p'(n)}$. (When $j = p'(n)$ this means we simply choose $s_{p'(n)} \leftarrow \{0,1\}^{p(n)}$ uniformly at random and output it.) The crucial observation here is that H_n^0 corresponds to outputting $G(s)$ for $s \leftarrow \{0,1\}^n$ chosen uniformly at random, while $H_n^{p'(n)}$ corresponds to outputting a $p(n)$-bit string chosen uniformly at random. Fixing any polynomial-time distinguisher D, this means that

$$\begin{aligned} \varepsilon(n) &\stackrel{\text{def}}{=} \left| \Pr_{s \leftarrow \{0,1\}^n}[D(G(s)) = 1] - \Pr_{r \leftarrow \{0,1\}^{p(n)}}[D(r) = 1] \right| \\ &= \left| \Pr_{s_{p'(n)} \leftarrow H_n^0}[D(s_{p'(n)}) = 1] - \Pr_{s_{p'(n)} \leftarrow H_n^{p'(n)}}[D(s_{p'(n)}) = 1] \right|. \qquad (6.7) \end{aligned}$$

Our goal is to prove that ε is negligible, implying that G is a pseudorandom generator.

Fix D as above, and consider the distinguisher D' that does the following when given a string $w \in \{0,1\}^{n+1}$ as input:

1. Choose $j \leftarrow \{1, \ldots, p'(n)\}$ uniformly at random.
2. Choose $\sigma_j \leftarrow \{0,1\}^{j-1}$ uniformly at random.
3. Set $s_j := (w, \sigma_j)$. Then run G starting from iteration $i = j + 1$ and compute $s_{p'(n)}$. Output $D(s_{p'(n)})$.

Analyzing the behavior of D' is more complicated than before, though the underlying ideas are the same. Fix n and say D' chooses $j = J$. If $w \leftarrow \{0,1\}^{n+1}$ was chosen uniformly at random, then s_J is uniformly distributed, and so the distribution on $s_{p'(n)}$ is exactly that of distribution H_n^J. That is,

$$\Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1 \mid j = J] = \Pr_{s_{p'(n)} \leftarrow H_n^J}[D(s_{p'(n)}) = 1].$$

Since each value for j is chosen with equal probability,

$$\begin{aligned}\Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] &= \frac{1}{p'(n)} \cdot \sum_{J=1}^{p'(n)} \Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1 \mid j = J] \\ &= \frac{1}{p'(n)} \cdot \sum_{J=1}^{p'(n)} \Pr_{s_{p'(n)} \leftarrow H_n^J}[D(s_{p'(n)}) = 1]. \qquad (6.8)\end{aligned}$$

On the other hand, say D' chooses $j = J$ and $w = \hat{G}(s)$ for $s \leftarrow \{0,1\}^n$ chosen uniformly at random. Mentally setting $s_{J-1} = (s, \sigma_J)$, we see that s_{J-1} is uniformly distributed and so the distribution on $s_{p'(n)}$ is now exactly that of distribution H_n^{J-1}. That is,

$$\Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1 \mid j = J] = \Pr_{s_{p'(n)} \leftarrow H_n^{J-1}}[D(s_{p'(n)}) = 1]$$

and then

$$\begin{aligned}\Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1] &= \frac{1}{p'(n)} \cdot \sum_{J=1}^{p'(n)} \Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1 \mid j = J] \\ &= \frac{1}{p'(n)} \cdot \sum_{J=1}^{p'(n)} \Pr_{s_{p'(n)} \leftarrow H_n^{J-1}}[D(s_{p'(n)}) = 1] \\ &= \frac{1}{p'(n)} \cdot \sum_{J=0}^{p'(n)-1} \Pr_{s_{p'(n)} \leftarrow H_n^J}[D(s_{p'(n)}) = 1]. \qquad (6.9)\end{aligned}$$

(Note that the indices of summation have been shifted in the final step.) We can now analyze how well D' distinguishes outputs of $\hat{G}$ from random:

$$\begin{aligned}
&\left| \Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1] - \Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] \right| \\
&= \frac{1}{p'(n)} \cdot \left| \sum_{J=0}^{p'(n)-1} \Pr_{s_{p'(n)} \leftarrow H_n^J}[D(s_{p'(n)}) = 1] - \sum_{J=1}^{p'(n)} \Pr_{s_{p'(n)} \leftarrow H_n^J}[D(s_{p'(n)}) = 1] \right| \\
&= \frac{1}{p'(n)} \cdot \left| \Pr_{s_{p'(n)} \leftarrow H_n^0}[D(s_{p'(n)}) = 1] - \Pr_{s_{p'(n)} \leftarrow H_n^{p'(n)}}[D(s_{p'(n)}) = 1] \right| \\
&= \frac{\varepsilon(n)}{p'(n)},
\end{aligned}$$

relying on Equations (6.8) and (6.9) for the first equality and Equation (6.7) for the final equality. (The second equality is due to the fact that the same terms are included in each sum, except for the first term of the left sum and the last term of the right sum.) Since $\hat{G}$ is a pseudorandom generator, D' runs in polynomial time, and p' is polynomial, we conclude that ε is negligible. ■

The hybrid technique. The *hybrid technique* is used in many proofs of security and is a basic tool for proving indistinguishability when a basic primitive is applied multiple times. The technique works by defining a series of hybrid distributions that bridge between two "extreme distributions", these being the distributions that we wish to prove indistinguishable (in the proof above, these correspond to the output of G and a random string, respectively). To apply the proof technique, three conditions should hold. First, the extreme hybrids should match the original cases of interest (in the proof above, this means that H_n^0 was equal to the distribution induced by G, while $H_n^{p'(n)}$ was equal to the uniform distribution). Second, it must be possible to translate the capability of distinguishing neighboring hybrids into the capability of breaking some underlying assumption (above, distinguishing H_n^i from H_n^{i+1} was essentially equivalent to distinguishing the output of $\hat{G}$ from random). Finally, the number of hybrids should be polynomial, so the "distinguishing success" is only reduced by a polynomial factor.

An explicit generator with arbitrary expansion factor. Let f be a one-way permutation with hard-core predicate hc. By combining the construction of Theorem 6.22 (that states that $\hat{G}(s) = (f(s), \mathsf{hc}(s))$ is a pseudorandom generator) with the proof of Theorem 6.23, we obtain that

$$G(s) = \left(f^{p'(n)}(s), \mathsf{hc}\left(f^{p'(n)-1}(s) \right), \ldots, \mathsf{hc}(s) \right)$$

is a pseudorandom generator with expansion factor $p(n) = n + p'(n)$. This generator is known as the Blum-Micali generator.

Modern stream cipher design. Many modern stream ciphers work by maintaining a *pseudorandom* internal state, much as in the construction of G shown above. In each iteration of the generator, some pseudorandom bits are output and the internal state is updated. The construction described in Theorem 6.23 (and in Figure 6.1) works in exactly this way, except that for a stream cipher only the σ_i values are output (and, in particular, $s'_{p'(n)}$ is not output) so that unbounded expansion can be achieved. The preceding proof validates this design principle for stream ciphers, since it shows that in some circumstances it can be used to achieve a provably-secure construction.

6.5 Constructing Pseudorandom Functions

Having shown how to construct pseudorandom generators from one-way permutations, we continue and show how to construct pseudorandom functions from pseudorandom generators. As defined in Section 3.6.1, a pseudorandom function is an efficiently-computable keyed function that looks random to any polynomial-time distinguisher; recall, this distinguisher receives oracle access to either a truly random function or a pseudorandom one.

We motivate the full construction by the following short example. Let G be a pseudorandom generator with expansion factor $\ell(n) = 2n$ (i.e., G is length doubling), and denote $G(s) = (G_0(s), G_1(s))$, where $|s| = |G_0(s)| = |G_1(s)| = n$. That is, the seed length is n, $G_0(s)$ denotes the first half of the output of $G(s)$, and $G_1(s)$ denotes the second half of the output. We now use G to construct a keyed function, using an n-bit key, that takes a *single bit* for input and outputs strings of length n. For a key k, define:

$$F_k(0) \stackrel{\text{def}}{=} G_0(k) \quad \text{and} \quad F_k(1) \stackrel{\text{def}}{=} G_1(k).$$

We claim that this function is pseudorandom.[2] This follows immediately from the fact that G is a pseudorandom generator: A random function mapping one bit to n bits is defined by a table of two n-bit values, each of which is chosen at random. Here, we have defined a keyed function by what is essentially a table of two n-bit values, each of which is *pseudorandom*. Thus, F_k (for randomly-chosen k) cannot be distinguished from a random function by any polynomial-time algorithm.

[2] Our formal definition of pseudorandom functions (Definition 3.23) assumes a length-preserving function having $\{0,1\}^n$ as its domain and range. The definition can be extended in the obvious way for functions having an arbitrary domain and range.

We now take this construction a step further and define a pseudorandom function that takes a *two-bit input*. For a key k, define:

$$F_k(00) = G_0(G_0(k)) \quad F_k(10) = G_0(G_1(k))$$
$$F_k(01) = G_1(G_0(k)) \quad F_k(11) = G_1(G_1(k)).$$

We now claim that the four strings $G_0(G_0(k))$, $G_0(G_1(k))$, $G_1(G_0(k))$, and $G_1(G_1(k))$ are pseudorandom, *even when viewed together*. (As above, this suffices to prove that the function F_k is pseudorandom.) Indeed, consider the following *hybrid distribution*:

$$G_0(k_0),\ G_1(k_0),\ G_0(k_1),\ G_1(k_1),$$

where $k_0, k_1 \leftarrow \{0,1\}^n$ are independent, uniformly-distributed strings. In this hybrid distribution, the random string k_0 takes the place of $G_0(k)$ and the random string k_1 takes the place of $G_1(k)$. Now, if it is possible to distinguish the hybrid distribution from the original distribution, then we would be able to distinguish between the pseudorandom string $G(k) = (G_0(k), G_1(k))$ and a truly random string (k_1, k_2), in contradiction to the pseudorandomness of G. Likewise, if it is possible to distinguish the hybrid distribution from a truly random string of length $4n$, then it would be possible to distinguish either $G(k_0) = (G_0(k_0), G_1(k_0))$ from a truly random string of length $2n$, or $G(k_1) = (G_0(k_1), G_1(k_1))$ from a truly random string of length $2n$. Once again, this contradicts the pseudorandomness of G. Combining the above, we have that the four strings are pseudorandom, and so the function defined is also pseudorandom. The formal proof of this fact is left as an exercise.

We can generalize the above constructions to obtain a pseudorandom function on n-bit inputs by defining

$$F_k(x) = G_{x_n}(\cdots G_{x_1}(k)\cdots),$$

where $x = x_1 \cdots x_n$; see Construction 6.24. The intuition for why this function is pseudorandom is the same as before, but the formal proof is now complicated by the fact that there are now exponentially-many values to consider (namely, $F_k(0\cdots0)$ through $F_k(1\cdots1)$).

CONSTRUCTION 6.24

Let G be a pseudorandom generator with expansion factor $\ell(n) = 2n$. Denote by $G_0(k)$ the first half of G's output, and by $G_1(k)$ the second half of G's output. For every $k \in \{0,1\}^n$, define the function $F_k : \{0,1\}^n \to \{0,1\}^n$ as:

$$F_k(x_1x_2\cdots x_n) = G_{x_n}\left(\cdots(G_{x_2}(G_{x_1}(k)))\cdots\right).$$

A pseudorandom function from a pseudorandom generator.

This construction can be viewed as a full binary tree of depth n, defined as follows (see Figure 6.2). The value at the root equals the key k. Then, for any node of value k', the left child of k' has value $G_0(k')$ and the right child of k' has value $G_1(k')$. The value of $F_k(x)$ for $x = x_1 \cdots x_n$ is then equal to the value at the *leaf* that is reached by traversing the tree according to x (that is, $x_i = 0$ means "go left in the tree", and $x_i = 1$ means "go right"). We stress that the function is only defined for inputs of length n, and thus only values in the leaves are ever output. The size of the tree is exponential in n; in particular, there are 2^n leaves. Nevertheless, to compute the function $F_k(x)$ we never need to construct and store the entire tree explicitly. Rather, we only need to compute the values on the path from the root to the appropriate leaf.

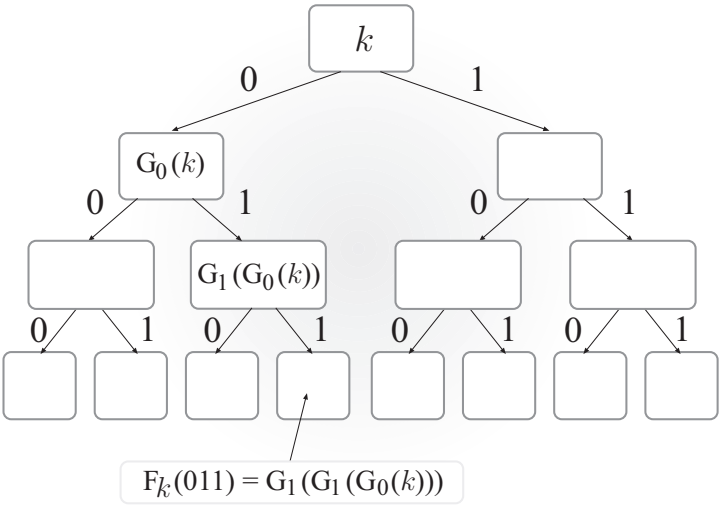

FIGURE 6.2: Constructing a pseudorandom function.

THEOREM 6.25 *If G is a pseudorandom generator with expansion factor $\ell(n) = 2n$, then Construction 6.24 is a pseudorandom function.*

PROOF (Sketch) The proof serves as another example of the hybrid technique. Let D be a probabilistic polynomial-time distinguisher that is given 1^n along with oracle access to a function that is either equal to a randomly-chosen function f mapping n-bit strings to n-bit strings, or is equal to F_k for a randomly-chosen key k.

We define a sequence of distributions on binary trees. By associating the values at the leaves of any given binary tree of depth n with strings of length n (as in Figure 6.2), we can equivalently think of these as being distributions over functions. Let H_n^i, for $0 \leq i \leq n$, denote the following distribution over binary trees of depth n: values for the nodes at level i are chosen independently

and uniformly at random from $\{0,1\}^n$; the value of any node at level $j \geq i+1$ is determined by looking at the value k' of this node's parent and setting the value of this node equal to $G_0(k')$ if it is a left child and setting the value equal to $G_1(k')$ if it is a right child. (Note that, viewing this as a function, values of nodes at levels 0 though $i-1$ are irrelevant.)

Notice that H_n^n is a truly random function mapping n-bit strings to n-bit strings, because the values of all the leaves (i.e., nodes at depth n) are chosen uniformly and independently at random. On the other hand, H_n^0 is exactly Construction 6.24 for a uniformly-chosen key, since only the root (at level 0) is chosen at random and the value of every other node in the tree is a deterministic function of the value of the root.

Using a hybrid argument as in the proof of Theorem 6.23, we obtain that if a polynomial-time distinguisher D can distinguish Construction 6.24 from a truly random function with non-negligible probability, then there must be values i for which H_n^i can be distinguished from H_n^{i+1} with non-negligible probability. We can use this to distinguish the pseudorandom generator from random. Intuitively this follows because the only difference between the neighboring hybrid distributions H_n^i and H_n^{i+1} is that in H_n^{i+1} the pseudorandom generator G is applied for one additional level on the way from the root to the leaves of the tree. The actual proof is trickier than this because we cannot hold the entire $(i+1)^{\text{th}}$ level of the tree (it may be exponential in size). Rather, let $t(n)$ be the maximum running-time of the distinguisher D who manages to distinguish Construction 6.24 from a random function. It follows that D makes at most $t(n)$ oracle queries to its oracle function. Now, let D' be a distinguisher for G that receives an input of length $2n \cdot t(n)$ that is either truly random or $t(n)$ invocations of $G(s)$ with independent random values of s each time. (Although we have not shown it here, it is not difficult to show that all of these samples together constitute a pseudorandom string of length $2n \cdot t(n)$.) Then, D' chooses a random $i \leftarrow \{0,\ldots,n-1\}$ and answers D's oracle queries as follows, initially holding an empty binary tree. Upon receiving a query $x = x_1 \cdots x_n$ from D, distinguisher D' uses $x_1 \cdots x_i$ to reach a node on the i^{th} level. Then, D' takes one of its input samples (of length $2n$) and labels the left child of the reached node with the first half of the sample and the right child with the second half of the sample. D' then continues to compute the output as in Construction 6.24. Note that in future queries, if the input x brings D' to a node that has already been filled, then D' answers consistently to the value that already exists there. Otherwise, D' uses a new sample from its input. (Notice that D' fills the tree *dynamically*, depending on D's queries. It does this because the full tree is too large to hold.)

The important observations are as follows:

1. If D' receives a truly random string of length $2n \cdot t(n)$, then it answers D' exactly according to the distribution H_n^{i+1}. This holds because all the values in level $i+1$ in the tree that are (dynamically) constructed by D' are random.

2. If D' receives pseudorandom input (i.e., $t(n)$ invocations of $G(s)$ with independent values of s each time), then it answers D' exactly according to H_n^i. This holds because the values in level $i+1$ are pseudorandom and generated by G, exactly as defined. (The seeds to these pseudorandom values are not known to D' but this makes no difference to the result.)

Using a hybrid analysis as in the proof of Theorem 6.23, we see that if D distinguishes Construction 6.24 from a truly random function with probability $\varepsilon(n)$, then D' distinguishes $t(n)$ invocations of $G(s)$ from a truly random string of length $2n \cdot t(n)$ with probability $\varepsilon(n)/n$. If $\varepsilon(n)$ is non-negligible, this contradicts the assumption that G is a pseudorandom generator. ■

6.6 Constructing (Strong) Pseudorandom Permutations

In this section, we show how pseudorandom permutations and strong pseudorandom permutations can be constructed from pseudorandom functions. Recall from Section 3.6.3 that a pseudorandom permutation is a pseudorandom function F that is also efficiently invertible (this implies that F_k is a permutation for any key k), while a *strong* pseudorandom permutation is additionally hard to distinguish from a random permutation even by an adversary given oracle access to both the permutation *and its inverse*.

Feistel networks revisited. A Feistel network, introduced in Section 5.2, is a way of constructing an invertible function from non-invertible operations. In some sense, this is exactly what we wish to do here. A Feistel network operates in a series of rounds. We view the input to the ith round as a string of length $2n$, divided into two n-bit halves L_{i-1} and R_{i-1} (the "left half" and the "right half", respectively). The output of the ith round will be the $2n$-bit string (L_i, R_i) where

$$L_i := R_{i-1} \quad \text{and} \quad R_i := L_{i-1} \oplus f_i(R_{i-1})$$

for some efficiently-computable (but not necessarily invertible) round function f_i mapping n-bit inputs to n-bit outputs.

Let us denote by $\mathsf{Feistel}_{f_1,\ldots,f_r}$ the r-round Feistel network using round functions $f_1, \ldots, f_r$. (That is, $\mathsf{Feistel}_{f_1,\ldots,f_r}(L_0, R_0)$ outputs the $2n$-bit string (L_r, R_r).) We saw in Section 5.2 that for every value of r, $\mathsf{Feistel}_{f_1,\ldots,f_r}$ is an efficiently-invertible permutation regardless of the round functions $\{f_i\}$. In particular, this means that if F is a pseudorandom function then $F^{(1)}$ defined as

$$F_k^{(1)}(x) \stackrel{\text{def}}{=} \mathsf{Feistel}_{F_k}(x)$$

is a keyed permutation. (Note that $F_k^{(1)}$ is a permutation over $\{0,1\}^{2n}$ when $k \in \{0,1\}^n$.) However, is $F^{(1)}$ pseudorandom?

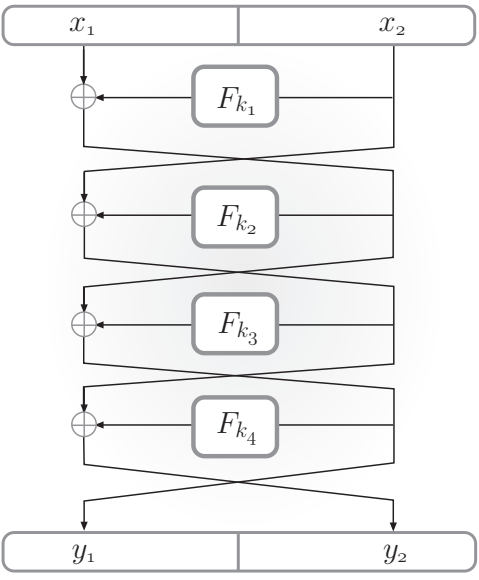

FIGURE 6.3: A four-round Feistel network, as used to construct a strong pseudorandom permutation from a pseudorandom function.

A little thought shows that $F^{(1)}$ is decidedly *not* pseudorandom. For any key $k \in \{0,1\}^n$, the first n bits of the output of $F_k^{(1)}$ (that is, L_1) are equal to the last n bits of the input (i.e., R_0), something that occurs with only negligible probability for a random function. Continuing in this vein, we can define a keyed permutation $F^{(2)} : \{0,1\}^{2n} \times \{0,1\}^{2n} \to \{0,1\}^{2n}$ as follows:

$$F^{(2)}_{k_1,k_2}(x) \stackrel{\text{def}}{=} \mathsf{Feistel}_{F_{k_1},F_{k_2}}(x). \tag{6.10}$$

(Note that k_1 and k_2 are independent keys.) Unfortunately, $F^{(2)}$ is not pseudorandom either, as you are asked to show in Exercise 6.18.

Given the above, it may be somewhat surprising that a *three*-round Feistel network *is* pseudorandom. That is, define the keyed function $F^{(3)}$, taking a key of length $3n$ and mapping $2n$-bit inputs to $2n$-bit outputs, as follows:

$$F^{(3)}_{k_1,k_2,k_3}(x) \stackrel{\text{def}}{=} \mathsf{Feistel}_{F_{k_1},F_{k_2},F_{k_3}}(x) \tag{6.11}$$

where, once again, k_1, k_2, and k_3 are chosen independently. It is possible to prove the following result:

THEOREM 6.26 *If F is a length-preserving pseudorandom function, then $F^{(3)}$ is a pseudorandom permutation that maps $2n$-bit strings to $2n$-bit strings (and uses a key of length $3n$).*

$F^{(3)}$ is not *strongly* pseudorandom (you are asked to demonstrate this in Exercise 6.19). Fortunately, adding a fourth round *does* yield a strong pseudorandom permutation. The details are given as Construction 6.27; see also Figure 6.3.

CONSTRUCTION 6.27

Let F be length-preserving, keyed function. Define the keyed permutation $F^{(4)}$ as follows:

- **Inputs:** A key $k \in \{0,1\}^{4n}$ parsed as $k = (k_1, k_2, k_3, k_4)$ with $|k_i| = n$, and an input $x \in \{0,1\}^{2n}$ parsed as (L_0, R_0) with $|L_0| = |R_0| = n$.
- **Computation:**
  1. Compute $L_1 := R_0$ and $R_1 := L_0 \oplus F_{k_1}(R_0)$.
  2. Compute $L_2 := R_1$ and $R_2 := L_1 \oplus F_{k_2}(R_1)$.
  3. Compute $L_3 := R_2$ and $R_3 := L_2 \oplus F_{k_3}(R_2)$.
  4. Compute $L_4 := R_3$ and $R_4 := L_3 \oplus F_{k_4}(R_3)$.
  5. Output (L_4, R_4).

A strong pseudorandom permutation from any pseudorandom function.

THEOREM 6.28 *If F is a length-preserving pseudorandom function, then Construction 6.27 is a strong pseudorandom permutation that maps $2n$-bit strings to $2n$-bit strings (and uses a key of length $4n$).*

The proofs of Theorems 6.26 and 6.28 are technical and are omitted, and we refer to [64] for those interested.

6.7 Necessary Assumptions for Private-Key Cryptography

Summing up what we have seen so far in this chapter:

1. If there exists a one-way permutation, then there exists a pseudorandom generator.
2. If there exists a pseudorandom generator, then there exists a pseudorandom function.
3. If there exists a pseudorandom function, then there exists a (strong) pseudorandom permutation.

Thus, pseudorandom generators and permutations can be achieved assuming the existence of one-way permutations. In actuality, it is possible to construct

pseudorandom generators from any one-way *function*, though we did not prove this here. In any case, we have the following fundamental theorem:

THEOREM 6.29 *If there exist one-way functions, then there exist pseudorandom generators, pseudorandom functions, and strong pseudorandom permutations.*

All of the private-key schemes that we have studied in Chapters 3 and 4 can be constructed from pseudorandom generators and pseudorandom functions. We therefore have:

THEOREM 6.30 *If there exists a one-way function, then there exists an encryption scheme that has indistinguishable encryptions under a chosen-ciphertext attack, and a message authentication code that is existentially unforgeable under a chosen message attack.*

Stated informally, *one-way functions are sufficient for all private-key cryptography.* Given this, we may wonder whether one-way functions are also *necessary.* In the rest of this section, we show that this is indeed the case.

Pseudorandomness implies one-way functions. We begin by showing that the existence of pseudorandom generators implies the existence of one-way functions:

PROPOSITION 6.31 *If there exists a pseudorandom generator, then there exists a one-way function.*

PROOF Let G be a pseudorandom generator with expansion factor of $\ell(n) = 2n$. (By Theorem 6.23, we know that the existence of a pseudorandom generator implies the existence of one with this expansion factor.) We show that G itself is one-way. Efficient computability is straightforward (since G can be computed in polynomial time). We show that the ability to invert G can be translated into the ability to distinguish the output of G from random. Intuitively, this holds because the ability to invert G implies the ability to find the seed used by the generator.

Let $\mathcal{A}$ be a probabilistic polynomial-time algorithm, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Invert}_{\mathcal{A},G}(n) = 1]$$

(cf. Definition 6.1). Construct the following distinguisher D that runs in polynomial time: on input a string $w \in \{0,1\}^{2n}$, run $\mathcal{A}(w)$ to obtain output x. If $G(x) = w$ then output 1; otherwise, output 0.

We now analyze the behavior of D. First consider the probability that D outputs 1 when its input string w is chosen at random. Since there are at

most 2^n values in the range of G (namely, the values $\{G(s)\}_{s\in\{0,1\}^n}$), the probability that w is in the range of G is at most 2^{-n}. When this is not the case, it is impossible for $\mathcal{A}$ to compute an inverse of w and thus impossible for D to output 1. We conclude that

$$\Pr_{w\leftarrow\{0,1\}^{2n}}[D(w)=1]\leq 2^{-n}.$$

On the other hand, if $w=G(s)$ for a seed $s\in\{0,1\}^n$ chosen uniformly at random, then, by definition, $\mathcal{A}$ computes a correct inverse (and so D outputs 1), with probability exactly $\varepsilon(n)$. We thus see that

$$\left|\Pr_{w\leftarrow\{0,1\}^{2n}}[D(w)=1]-\Pr_{s\leftarrow\{0,1\}^{n}}[D(G(s))=1]\right|\geq\varepsilon(n)-\frac{1}{2^n}\,.$$

Since G is a pseudorandom generator, there exists a negligible function negl for which $\varepsilon(n)-\frac{1}{2^n}\leq\mathsf{negl}(n)$. We conclude that $\varepsilon(n)$ is negligible, proving that G is a one-way function. ■

Private-key encryption schemes imply one-way functions. Proposition 6.31 tells us that if we want to build pseudorandom generators or functions, then we need to assume that one-way functions exist. This does *not* immediately imply that one-way functions are needed for constructing secure private-key encryption schemes, since it may be possible to construct secure encryption schemes without relying on these primitives. Furthermore, it *is* possible to construct perfectly-secret encryption schemes (see Chapter 2), as long as the plaintext is no longer than the key. Thus, the proof that secure private-key encryption implies one-way function must be more subtle.

We now prove that an encryption scheme satisfying the weakest definition of security we have considered (namely, a scheme having indistinguishable encryptions in the presence of an eavesdropper) implies the existence of a one-way function.

PROPOSITION 6.32 *If there exists a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper (as in Definition 3.8), then there exists a one-way function.*

PROOF We rely in the proof on the fact that Definition 3.8 requires security to hold for the encryption of arbitrary-length messages. Actually, all we need is for the encryption scheme to support the encryption of messages longer than the key. Importantly, the theorem does *not* hold for encryption schemes (such as the perfectly-secure one-time pad) that encrypt messages of length equal to the key.

Let $\Pi=(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$ be a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper. Assume that

when an n-bit key is used, Enc uses at most $\ell(n)$ bits of randomness in order to encrypt a plaintext message of length $2n$. Denote an encryption of a message m with key k and random coins r by $\mathsf{Enc}_k(m;r)$.

Define a function f by

$$f(k,m,r) \stackrel{\text{def}}{=} (\mathsf{Enc}_k(m;r), m),$$

where $|k| = n$, $|m| = 2n$, and $|r| = \ell(n)$. We claim that f is a one-way function. The fact that it can be efficiently computed is immediate. We show that it is hard to invert. Let $\mathcal{A}$ be a probabilistic polynomial-time algorithm and set

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1]$$

(cf. Definition 6.1). We show that $\varepsilon(n)$ is negligible, which will complete the proof that f is one-way.

Consider the following probabilistic polynomial-time adversary $\mathcal{A}'$ that runs in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n)$:

Adversary $\mathcal{A}'(1^n)$

1. Choose random $m_0, m_1 \leftarrow \{0,1\}^{2n}$ and output these two messages. Receive in return a challenge ciphertext c.
2. Run $\mathcal{A}(c, m_0)$ to obtain (k', m', r'). If $f(k', m', r') = (c, m_0)$, output 0; else, output a random bit.

Let us analyze the probability that $\mathcal{A}'$ outputs 0 when $b = 0$. (Recall that $b = 0$ means that the challenge ciphertext is an encryption of m_0.) Let $\mathsf{invert}_{\mathcal{A}}$ denote the event that $\mathcal{A}$ outputs (k', m', r') with $f(k', m', r') = (c, m_0)$. (When $\mathsf{invert}_{\mathcal{A}}$ occurs, the key k' output by $\mathcal{A}$ may not be the "correct key" — i.e., it may not be equal to the key k used by the experiment to compute the challenge ciphertext — but this does not matter for our purposes.) Observe that when $b = 0$ the event $\mathsf{invert}_{\mathcal{A}}$ occurs with probability exactly $\varepsilon(n)$. This is true since the key k used to compute c is chosen uniformly at random, as are the message m_0 and the random coins used to compute c.

When $\mathsf{invert}_{\mathcal{A}}$ occurs, adversary $\mathcal{A}'$ outputs 0. When $\mathsf{invert}_{\mathcal{A}}$ does not occur, $\mathcal{A}'$ outputs a random bit. So, the probability that $\mathcal{A}'$ succeeds (i.e., outputs the correct answer) when $b = 0$ is given by

$$\begin{aligned}
&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1 \mid b = 0\right] \\
&\quad = \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 0\right] + \frac{1}{2} \cdot \left(1 - \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 0\right]\right) \\
&\quad = \varepsilon(n) + \frac{1}{2} \cdot (1 - \varepsilon(n)) \\
&\quad = \frac{1}{2} + \frac{\varepsilon(n)}{2}.
\end{aligned}$$

We proceed to analyze the probability that $\mathcal{A}'$ outputs 1 when $b = 1$. As before, we begin by determining the probability that $\mathsf{invert}_{\mathcal{A}}$ occurs. At first

sight, it may appear that $\mathsf{invert}_{\mathcal{A}}$ can never occur when $b = 1$ since then c is an encryption of m_1 and so, seemingly, $\mathcal{A}$ cannot possibly find (k', m', r') with $f(k', m', r') = (c, m_0)$. This is not true, however, since for some $c = \mathsf{Enc}_k(m_1)$ there may exist a different key k' such that $m_0 = \mathsf{Dec}_{k'}(c)$; indeed perfectly-secret encryption schemes always have this property for every m_0 and m_1.

Nevertheless, we show that when $b = 1$ the event $\mathsf{invert}_{\mathcal{A}}$ occurs with at most negligible probability. To see this, fix a challenge ciphertext $c = \mathsf{Enc}_k(m_1)$ and note that when $b = 1$ this ciphertext is independent of m_0. Now, there are at most 2^n possible messages — one for each possible value of the key — that the ciphertext c can correspond to. If m_0 happens to be one of these possibilities, then we cannot bound the probability that $\mathsf{invert}_{\mathcal{A}}$ occurs. On the other hand, if m_0 is *not* one of these possibilities, then $\mathsf{invert}_{\mathcal{A}}$ cannot possibly occur (because in this case (c, m_0) is not in the range of f). Since there are at most 2^n possible messages corresponding to c, and m_0 is chosen uniformly at random from $\{0,1\}^{2n}$, the probability that $\mathsf{invert}_{\mathcal{A}}$ occurs is at most $2^n/2^{2n} = 2^{-n}$. This, in turn, means that the probability that $\mathcal{A}'$ succeeds when $b = 1$ is given by

$$\begin{aligned}\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1 \mid b = 1\right] &= \frac{1}{2} \cdot \left(1 - \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 1\right]\right) \\ &\geq \frac{1}{2} \cdot (1 - 2^{-n}) \\ &= \frac{1}{2} - 2^{-(n+1)}.\end{aligned}$$

Putting the above together along with the fact that b is chosen at random, we have:

$$\begin{aligned}&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1\right] \\ &= \frac{1}{2} \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1 \mid b = 0\right] + \frac{1}{2} \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1 \mid b = 1\right] \\ &\geq \frac{1}{2} \cdot \left(\frac{1}{2} + \frac{\varepsilon(n)}{2}\right) + \frac{1}{2} \cdot \left(\frac{1}{2} - 2^{-(n+1)}\right) \\ &= \frac{1}{2} + \frac{\varepsilon(n)}{4} - \frac{1}{2^{n+2}}.\end{aligned}$$

Security of Π means that $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)$ for some negligible function negl. This in turn implies that $\varepsilon(n)$ is negligible, completing the proof that f is one-way. ∎

Message authentication codes imply one-way functions. It is also true that message authentication codes satisfying Definition 4.2 imply the existence of one-way functions. As in the case of private-key encryption, the proof of this fact is somewhat subtle because unconditionally-secure message authentication codes *do* exist when there is an *a priori* bound on the number

of messages that will be authenticated. Thus, a proof relies on the fact that Definition 4.2 requires security even when the adversary sees the authentication tags of an *arbitrary* (polynomial) number of messages. The proof is rather involved, so we do not give it here.

Discussion. We conclude that the existence of one-way functions is both a necessary and sufficient assumption for achieving all non-trivial private-key cryptography. In other words, the assumption regarding the existence of one-way functions is *minimal* as far as private-key cryptography is concerned. This seems not to be the case for public-key encryption that we will study later. Although one-way functions are necessary also for public-key encryption, they appear not to be sufficient. (Besides the fact that we do not know how to construct public-key encryption from one-way functions, there is also evidence that such constructions are, in some sense, "unlikely to exist".)

6.8 A Digression – Computational Indistinguishability

The notion of computational indistinguishability is central to the theory of cryptography. It underlies much of what we have seen in this chapter, and is therefore worthy of explicit treatment. Informally speaking, two probability distributions are computationally indistinguishable if no efficient algorithm can tell them apart (or *distinguish* them). This is formalized as follows. Let D be some probabilistic polynomial-time algorithm, or distinguisher. Then, D is provided either with a sample from the first distribution or the second one. We say that the distributions are *computationally indistinguishable* if every such distinguisher D outputs 1 with almost the same probability upon receiving a sample from the first or second distribution. This should sound very familiar, and is in fact exactly how we defined pseudorandom generators. Indeed, a pseudorandom generator is an algorithm that generates a distribution that is computationally indistinguishable from the uniform distribution over strings of a certain length. Below, we will formally redefine the notion of a pseudorandom generator in this way.

The actual definition of computational indistinguishability refers to *probability ensembles*. These are infinite sequences of probability distributions (rather than being a single distribution). This formalism is a necessary consequence of the asymptotic approach, because distinguishing two fixed finite distributions is "easy" using exhaustive search.

DEFINITION 6.33 *Let I be a countable set. A* probability ensemble indexed by *I is a collection of random variables $\{X_i\}_{i \in I}$.*

In most cases, I is either the natural numbers $\mathbb{N}$ or an efficiently computable subset of $\{0,1\}^*$. When $I = \mathbb{N}$, an ensemble is just a sequence of random variables $X_1, X_2, ...$ and the random variable X_n might correspond to the output of some cryptographic scheme when the security parameter is set to n. In this case X_n would typically take values in $\{0,1\}^{\leq p(n)}$ (i.e., bit-strings of length at most $p(n)$) for some polynomial p.

With this notation in hand, we can now formally define what it means for two ensembles to be computationally indistinguishable.

DEFINITION 6.34 *Two probability ensembles $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ are* computationally indistinguishable*, denoted $X \stackrel{c}{\equiv} Y$, if for every probabilistic polynomial-time distinguisher D there exists a negligible function* negl *such that:*

$$|\Pr[D(1^n, X_n) = 1] - \Pr[D(1^n, Y_n) = 1]| \leq \mathsf{negl}(n),$$

where the notation $D(1^n, X_n)$ means that x is chosen according to distribution X_n and then $D(1^n, x)$ is run.

The distinguisher D is given the unary input 1^n so that it can run in time polynomial in n. This is important when the output of X_n and Y_n may be very short.

6.8.1 Pseudorandomness and Pseudorandom Generators

Pseudorandomness is just a special case of computational indistinguishability. Let $U_{\ell(n)}$ denote the uniform distribution over $\{0,1\}^{\ell(n)}$. Then we have the following definition:

DEFINITION 6.35 *An ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ is* pseudorandom *if for some polynomial ℓ, the ensemble X is computationally indistinguishable from the ensemble $U = \{U_{\ell(n)}\}_{n\in\mathbb{N}}$.*

This, in turn, can be used to redefine the notion of a pseudorandom generator (cf. Definition 3.14):

DEFINITION 6.36 *Let $\ell(\cdot)$ be a polynomial and let G be a (deterministic) polynomial-time algorithm where for all s it holds that $|G(s)| = \ell(|s|)$. We say that G is a* pseudorandom generator *if the following two conditions hold:*

1. (Expansion:) *For every n it holds that $\ell(n) > n$.*
2. (Pseudorandomness:) *The ensemble $\{G(U_n)\}_{n\in\mathbb{N}}$ is pseudorandom.*

Many of the other definitions and assumptions in this book can also be cast as special cases of computational indistinguishability. Despite the fact that this involves jumping ahead, we give one example: the decisional Diffie-Hellman (DDH) assumption of Section 7.3.2 can be formalized by stating that the ensemble of tuples of the type $(\mathbb{G}, q, g, g^x, g^y, g^{xy})$ is *computationally indistinguishable* from the ensemble of tuples of the type $(\mathbb{G}, q, g, g^x, g^y, g^z)$, where $(\mathbb{G}, q, g)$ are output by some algorithm $\mathcal{G}(1^n)$ and x, y, z are randomly chosen from $\mathbb{Z}_q$.

6.8.2 Multiple Samples

An important general theorem regarding computational indistinguishability is that multiple samples of computationally indistinguishable ensembles are also computationally indistinguishable. For example, consider a pseudorandom generator G with expansion factor ℓ. Then, the output of $p(n)$ independent applications of G is a pseudorandom string of length $p(n) \cdot \ell(n)$. That is, $\{(G(s_1), \ldots G(s_{p(n)}))\}$ is computationally indistinguishable from the uniform distribution over $\{0,1\}^{p(n)\ell(n)}$, where $s_1, \ldots, s_{p(n)}$ are independently chosen random strings of length n. We prove this theorem because it is used very often in cryptographic proofs (e.g., we relied on it in the proof of Theorem 6.25), and also because it is another example of a hybrid argument (as seen in the proof of Theorem 6.23).

Say an ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ is efficiently sampleable if there exists a probabilistic polynomial-time algorithm S such that for every n, the random variables $S(1^n)$ and X_n are identically distributed. That is, the algorithm S is an efficient way of sampling X. Clearly, the ensemble generated by a pseudorandom generator is efficiently sampleable: the algorithm S chooses a random string s of length n and then outputs $G(s)$. We now prove that if two *efficiently sampleable* ensembles X and Y are computationally indistinguishable, then a polynomial number of (independent) samples of X are computationally indistinguishable from a polynomial number of (independent) samples of Y. (The theorem does not hold if X and Y are not efficiently sampleable.) We denote by $\overline{X} = \{(X_n^{(1)}, \ldots, X_n^{(p(n))})\}_{n\in\mathbb{N}}$ the ensemble generated by $p(n)$ independent samples of X_n; likewise for $\overline{Y}$. For the sake of clarity, we do not explicitly give the distinguisher the input 1^n, but assume that it knows the value of the security parameter and can run in time polynomial in n.

THEOREM 6.37 *Let X and Y be efficiently sampleable ensembles that are computationally indistinguishable. Then, for every polynomial $p(\cdot)$, the ensemble $\overline{X} = \{(X_n^{(1)}, \ldots, X_n^{(p(n))})\}_{n\in\mathbb{N}}$ is computationally indistinguishable from the ensemble $\overline{Y} = \{(Y_n^{(1)}, \ldots, Y_n^{(p(n))})\}_{n\in\mathbb{N}}$.*

PROOF The proof is by reduction. We show that if there exists a probabilistic polynomial-time distinguisher D that distinguishes $\overline{X}$ from $\overline{Y}$ with non-negligible success, then there exists a probabilistic polynomial-time distinguisher D' that distinguishes a single sample of X from a single sample of Y with non-negligible success. Formally, fix some probabilistic polynomial-time distinguisher D and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \left| \Pr\left[D(X_n^{(1)}, \ldots, X_n^{(p(n))}) = 1\right] - \Pr\left[D(Y_n^{(1)}, \ldots, Y_n^{(p(n))}) = 1\right] \right|.$$

For $0 \leq i \leq p(n)$, we define a *hybrid* random variable H_n^i as a sequence containing i independent copies of X_n followed by $p(n)-i$ independent copies of Y_n. I.e.,

$$H_n^i \stackrel{\text{def}}{=} \left(X_n^{(1)}, \ldots, X_n^{(i)}, Y_n^{(i+1)}, \ldots, Y_n^{(p(n))}\right).$$

Notice that $H_n^0 = \overline{Y}_n$ and $H_n^{p(n)} = \overline{X}_n$. The main idea behind the hybrid argument is that if D can distinguish these extreme hybrids, then it can also distinguish neighboring hybrids (even though it was not "designed" to do so). In order to see this, and before we proceed to the formal argument, we present the basic hybrid analysis. We know that:

$$\begin{aligned} &\left|\Pr[D(\overline{X}_n) = 1] - \Pr[D(\overline{Y}_n) = 1]\right| \\ &\quad = \left| \sum_{i=0}^{p(n)-1} \Pr[D(H_n^i) = 1] - \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1] \right|. \end{aligned}$$

This follows from the fact that the only terms remaining in this telescopic sum are $\Pr[D(H_n^0) = 1]$ and $\Pr[D(H_n^{p(n)}) = 1]$. Therefore,

$$\begin{aligned} \varepsilon(n) &= \left|\Pr[D(\overline{X}_n) = 1] - \Pr[D(\overline{Y}_n) = 1]\right| \\ &= \left| \sum_{i=0}^{p(n)-1} \Pr[D(H_n^i) = 1] - \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1] \right| \\ &\leq \sum_{i=0}^{p(n)-1} \left|\Pr[D(H_n^i) = 1] - \Pr[D(H_n^{i+1}) = 1]\right|. \end{aligned}$$

Thus, there exists an i for which

$$\left|\Pr[D(H_n^i) = 1] - \Pr[D(H_n^{i+1}) = 1]\right| \geq \varepsilon(n)/p(n).$$

The only difference between H_n^i and H_n^{i+1} occurs in the $(i+1)$st sample (in both distributions, the first i samples are from X_n and the last $n-i-1$ samples are from Y_n). So, the fact that D can distinguish between H_n^i and H_n^{i+1} can be used to construct a D' that distinguishes between a single sample of X and a single sample of Y, in contradiction to the assumption that $X \stackrel{\text{c}}{\equiv} Y$. In

the formal proof below, D' will choose i at random (because it does not know for which values of i it holds that D distinguishes well).

Formally, we construct a probabilistic polynomial-time distinguisher D' for a single sample of X_n and Y_n. Upon input a single sample α, D' chooses a random $i \leftarrow \{0, \ldots, p(n)-1\}$, generates the vector $\overline{H}_n = (X_n^{(1)}, \ldots, X_n^{(i)}, \alpha, Y_n^{(i+2)}, \ldots, Y_n^{(p(n))})$, invokes D on the vector $\overline{H}_n$, and outputs whatever D does.[3] Now, if α is distributed according to X_n, then $\overline{H}_n$ is distributed exactly like H_n^{i+1} (because the first $i+1$ samples are from X_n and the last $n-i-1$ from Y_n). In contrast, if α is distributed according to Y_n, then $\overline{H}_n$ is distributed exactly like H_n^i (because the first i samples are from X_n and the last $n-i$ from Y_n). This argument holds because the samples are independent and so it makes no difference who generates the samples and in which order. Now, each i is chosen with probability exactly $1/p(n)$. Therefore,

$$\Pr[D'(X_n) = 1] = \frac{1}{p(n)} \cdot \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1]$$

and

$$\Pr[D'(Y_n) = 1] = \frac{1}{p(n)} \cdot \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i}) = 1].$$

It therefore follows that:

$$\begin{aligned}
&|\Pr[D'(X_n) = 1] - \Pr[D'(Y_n) = 1]| \\
&= \frac{1}{p(n)} \cdot \left| \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1] - \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i}) = 1] \right| \\
&= \frac{1}{p(n)} \cdot \left| \Pr[D(H_n^{p(n)}) = 1] - \Pr[D(H_n^{0}) = 1] \right| \\
&= \frac{1}{p(n)} \cdot \left| \Pr[D(\overline{X}_n) = 1] - \Pr[D(\overline{Y}_n) = 1] \right| \;\geq\; \frac{\varepsilon(n)}{p(n)}.
\end{aligned}$$

Since $X \stackrel{c}{\equiv} Y$, we know that there exists a negligible function negl such that $|\Pr[D'(X_n) = 1] - \Pr[D'(Y_n) = 1]| \leq \mathsf{negl}(n)$. Since p is polynomial, this implies that ε must be negligible. ∎

[3] The efficient sampleability of X and Y is needed for constructing the vector $\overline{H}_n$.

References and Additional Reading

The notion of a one-way function was first suggested by Diffie and Hellman [47] and was later formalized and studied by Yao [149]. The concept of hard-core bits was introduced by Blum and Micali [23], and the fact that there exists a hard-core bit for every one-way function was proved by Goldreich and Levin [68].

The notion of pseudorandomness was introduced first by Yao [149] and the first construction of pseudorandom generators (under a specific number-theoretic hardness assumption) was given by Blum and Micali [23]. The construction of a pseudorandom generator from any one-way permutation was given by Yao [149], and the fact that pseudorandom generators can be constructed from any one-way function was shown by Håstad et al. [74]. Pseudorandom functions were defined and constructed by Goldreich, Goldwasser and Micali [67] and their extension to pseudorandom permutations was presented by Luby and Rackoff [97]. Goldreich's book [64] has a very clear and concise proof of Theorem 6.28. The fact that one-way functions are a necessary assumption for most of private-key cryptography was shown in [79].

Most of the presentation in this chapter follows Goldreich's book [64]. We highly recommend this book to students who are interested in furthering their understanding of the foundations of cryptography. This chapter is only a taste of the rich theory that cryptography has to offer.

Exercises

6.1 Show that the addition function $f(x, y) = x + y$ (where $|x| = |y|$ and x and y are interpreted as natural numbers) is not one-way. Likewise, show that $f(x) = x^2$ is not one-way.

6.2 Prove that if there exists a one-way function, then there exists a one-way function f such that for every n, $f(0^n) = 0^n$. Provide a full (formal) proof of your answer. Note that this demonstrates that for infinitely many values x, the function f is easy to invert. Why does this not contradict one-wayness?

6.3 Show that if there exists a one-way function, then there exists a length-preserving one-way function. Provide a full proof of your answer.

> **Hint:** Let f be a one-way function and let $p(\cdot)$ be a polynomial such that $|f(x)| \leq p(|x|)$ (justify the existence of such a p). Define $f'(x) = f(x)\|10^{p(|x|)-|f(x)|}$. Prove that f' is length-preserving and one-way.

6.4 Prove that if f is a one-way function, then $g(x_1, x_2) = (f(x_1), x_2)$ where $|x_1| = |x_2|$ is also a one-way function. Observe that g fully reveals half of its input bits, but is nevertheless still one-way.

6.5 Let f be a length-preserving one-way function, and let hc be a hard-core predicate for f. Define G as $G(x) = (f(x), \mathsf{hc}(x))$. Is G a pseudorandom generator? Prove your answer.

6.6 Prove that there exist one-way functions if and only if there exist families of one-way functions. Discuss why your proof does not carry over to the case of one-way permutations.

6.7 Let f be a one-way function. Is $g(x) = f(f(x))$ necessarily a one-way function? What about $g(x) = (f(x), f(f(x)))$? Prove your answers.

6.8 This exercise is for students who have taken a course in complexity theory or are otherwise familiar with $\mathcal{NP}$ completeness.

(a) Show that the existence of one-way functions implies $\mathcal{P} \neq \mathcal{NP}$.

(b) Assume that $\mathcal{P} \neq \mathcal{NP}$. Show that there exists a function f that is: **(1)** computable in polynomial time, **(2)** hard to invert *in the worst case* (i.e., for all probabilistic polynomial-time $\mathcal{A}$, $\Pr_{x \leftarrow \{0,1\}^n}[f(\mathcal{A}(f(x))) = f(x)] \neq 1$), but **(3)** is *not* one-way.

6.9 Let $x \in \{0,1\}^n$ and denote $x = x_1 \cdots x_n$. Prove that if there exists a one-way function, then there exists a one-way function f such that for every i there exists an algorithm A_i such that

$$\Pr_{x \leftarrow \{0,1\}^n}[A_i(f(x)) = x_i] \geq \frac{1}{2} + \frac{1}{2n}\,.$$

(This exercise demonstrates that it is not possible to claim that every one-way function hides at least one *specific* bit of the input.)

6.10 Show that if a one-to-one function has a hard-core predicate, then it is one-way.

6.11 Complete the proof of Proposition 6.15 by finding the Chernoff bound and applying it to the improved procedure of $\mathcal{A}'$ for guessing x_i.

6.12 Prove Claim 6.21.

6.13 Let G be a pseudorandom generator. Prove that

$$G'(x_1 \| \cdots \| x_n) = \big(G(x_1) \| G(x_2) \| \cdots \| G(x_n)\big)\,,$$

where $|x_1| = \cdots = |x_n| = n$, is a pseudorandom generator.

Hint: Use a hybrid argument. You may not use Theorem 6.37.

6.14 Prove that the function G' defined by

$$G'(s) = G_0(G_0(s)), G_0(G_1(s)), G_1(G_0(s)), G_1(G_1(s))$$

is a pseudorandom generator with expansion factor $\ell(n) = 4n$.

6.15 Show that if Construction 6.24 is modified so that the adversary is allowed to query $F_k(x)$ for any string $x \in \{0,1\}^{1 \leq n}$ (i.e., any non-empty string of length at most n), then the construction is no longer a pseudorandom function.

6.16 Prove that if there exists a pseudorandom function F that, using a key of length n, maps $p(n)$-bit inputs to single-bit outputs, then there exists a pseudorandom function that maps $p(n)$-bit inputs to n-bit outputs. (Here n, as usual, denotes the security parameter.) You should give a direct construction that does not rely on the results of Section 6.7.

Hint: Use a key of length n^2, and prove your construction secure using a hybrid argument.

6.17 Assuming the existence of a pseudorandom permutation, prove that there exists a keyed permutation F that is pseudorandom but is *not* strongly pseudorandom.

Hint: Though this follows from Exercise 6.19, a direct proof is possible.

6.18 Prove that a two-round Feistel network using pseudorandom round functions (as in Equation (6.10)) is not pseudorandom.

6.19 Prove that a three-round Feistel network using pseudorandom round functions (as in Equation (6.11)) is not *strongly* pseudorandom.

Hint: This is significantly more difficult than the previous exercise. Use a distinguisher that makes two queries to the permutation and one query to its inverse.

6.20 Let G be a pseudorandom function with expansion factor $\ell(n) = n + 1$. Prove that G is a one-way function.

6.21 Let $X = \{X_n\}_{n \in \mathbb{N}}$ and $Y = \{Y_n\}_{n \in \mathbb{N}}$ be computationally indistinguishable probability ensembles.

(a) Prove that for any probabilistic polynomial-time algorithm $\mathcal{A}$ it holds that $\{\mathcal{A}(X_n)\}_{n \in \mathbb{N}}$ and $\{\mathcal{A}(Y_n)\}_{n \in \mathbb{N}}$ are computationally indistinguishable.

(b) Prove that the above may no longer hold if $\mathcal{A}$ does not run in polynomial time.

Part III

Public-Key (Asymmetric) Cryptography

Chapter 7

Number Theory and Cryptographic Hardness Assumptions

Modern cryptography, as we have seen, is almost always based on an assumption that *some* problem cannot be solved in polynomial time. (See Section 1.4.2 for a discussion of this point.) In Chapters 3 and 4, for example, we saw that efficient private-key cryptography — both encryption and message authentication — can be based on the assumption that pseudorandom permutations exist. Recall that, roughly speaking, this means that there exists some keyed permutation F for which it is impossible to distinguish in polynomial time between interactions with F_k (for a randomly-chosen key k) and interactions with a truly random permutation.

On the face of it, the assumption that pseudorandom permutations exist seems quite strong and unnatural, and it is reasonable to ask whether this assumption is likely to be true or whether there is any evidence to support it. In Chapter 5 we explored how pseudorandom permutations (i.e., block ciphers) are constructed in practice. The resistance of these constructions to attack at least serves as an indication that the existence of pseudorandom permutations is plausible. Still, it is difficult to imagine looking at some F and somehow being convinced on any intuitive level that it is a pseudorandom permutation. Moreover, the current state of our theory is such that we do not know how to prove the pseudorandomness of any of the existing practical constructions relative to any "more reasonable" assumption. All in all, this is a not entirely satisfying state of affairs.

In contrast, as mentioned in Chapter 3 (and investigated in detail in Chapter 6) it is possible to *prove* that pseudorandom permutations exist based on the much milder assumption that one-way functions exist. (Informally, a function is *one-way* if it is easy to compute but hard to invert; see Section 7.4.1.) Apart from a brief discussion in Section 6.1.2, however, we have not yet seen any concrete examples of functions believed to be one-way.

One of the goals of this chapter is to introduce various problems that are believed to be "hard", and to present the conjectured one-way functions that can be based on these problems.[1] The second goal of this chapter is to develop

[1] Recall that we currently do not know how to *prove* that one-way functions exist, and so the best we can do is to base one-way functions on assumptions regarding the hardness of certain problems.

the basis needed for studying public-key cryptography (the next main topic of this book).

All the examples we explore will be *number-theoretic* in nature, and we therefore begin with a short introduction to number theory and group theory. Because we are additionally interested in problems that can be solved efficiently (even a one-way function needs to be easy to compute in one direction, and a cryptographic scheme must admit efficient algorithms for the honest parties), we also initiate a study of *algorithmic* number theory. Thus, even the reader who is familiar with number theory or group theory is encouraged to read this chapter, since algorithmic aspects are typically ignored in a purely mathematical treatment of these topics.

In the context of algorithmic number theory, a brief word is in order regarding what is meant by "polynomial time". An algorithm's running time is always measured as a function of the length(s) of its input(s). (If the algorithm is given as additional input a security parameter 1^n then the total input length is increased by n.) This means, for example, that the running time of an algorithm taking as input an integer N is measured in terms of $\|N\|$, the *length of the binary representation of* N, and not in terms of N itself. An algorithm running in time $\Theta(N)$ on input N is thus actually an *exponential*-time algorithm when measured in terms of its input length $\|N\| = \Theta(\log N)$.

The material in this chapter is not intended to be a comprehensive survey of number theory, but is intended rather to present the minimal amount of material needed for the cryptographic applications discussed in the remainder of the book. Accordingly, our discussion of number theory is broken into two: the material covered in this chapter is sufficient for understanding Chapters 8–10, 12, and 13. In Chapter 11, additional number theory is developed that is used only within that chapter.

The reader may be wondering why there was no discussion of number theory thus far, and why it is suddenly needed now. There are two reasons for placing number theory at this point of the book:

1. This chapter can be viewed as a culmination of the "top down" approach we have taken in developing private-key cryptography in Chapters 3–6. That is, we have shown in Chapters 3 and 4 that all of private-key cryptography can be based on pseudorandom functions and permutations. The latter can be instantiated in practice using block ciphers, as explored in Chapter 5, but can also be constructed in a rigorous and provably-sound manner from any one-way function, as shown in Chapter 6. Here, we take this one step further and show how one-way functions can be based on certain hard mathematical problems. We summarize this top-down approach in Figure 7.1.

2. A second motivation for studying this material illustrates a difference between the private-key setting we have been concerned with until now, and the *public-key* setting with which we will be concerned in the remainder of the book. (The public-key setting will be introduced in

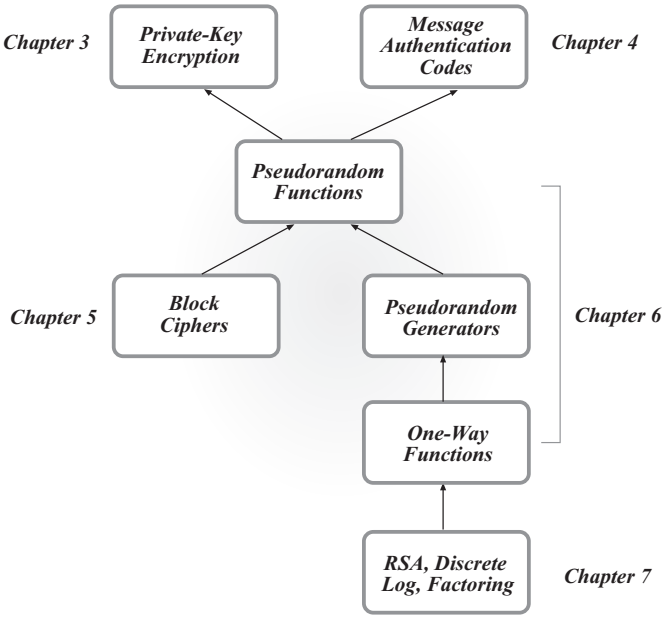

FIGURE 7.1: The world of private-key cryptography: a top-down approach (arrows represent implication).

Chapter 9.) Namely, in the private-key setting there exist suitable primitives (i.e., hash functions and pseudorandom generators, functions, and permutations) for constructing schemes, and these primitives can be constructed efficiently — at least in a heuristic sense — without invoking any number theory. In the public-key setting, however, *all known efficient constructions rely on hard mathematical problems from algorithmic number theory.* (We will also study constructions that do not rely directly on number theory. Unfortunately, however, these are far less efficient.)

The material in this chapter thus serves as both a culmination of what we have studied so far in private-key cryptography, as well as the foundation upon which public-key cryptography stands.

7.1 Preliminaries and Basic Group Theory

We begin with a review of prime numbers and basic modular arithmetic. Even the reader who has seen these topics before should skim the next two

sections since some of the material may be new and we include proofs for most of the stated results. (Any omitted proofs can be found in standard algebra texts; see the references at the end of this chapter.)

7.1.1 Primes and Divisibility

The set of integers is denoted by $\mathbb{Z}$. For $a, b \in \mathbb{Z}$, we say that a *divides* b, written $a \mid b$, if there exists an integer c such that $ac = b$. If a does not divide b, we write $a \nmid b$. (We are primarily interested in the case where a, b and c are all positive, though the definition makes sense even when one or more of these is negative or zero.) A simple observation is that if $a \mid b$ and $a \mid c$ then $a \mid (Xb + Yc)$ for any $X, Y \in \mathbb{Z}$.

If $a \mid b$ and a is positive, we call a a *divisor* of b. If in addition $a \notin \{1, b\}$ then a is called a *non-trivial* divisor, or a *factor*, of b. A positive integer $p > 1$ is *prime* if it has no factors; i.e., it has only two divisors: 1 and itself. A positive integer greater than 1 that is not prime is called *composite*. By convention, '1' is neither prime nor composite.

A fundamental theorem of arithmetic is that every integer greater than 1 can be expressed *uniquely* (up to ordering) as a product of primes. That is, any positive integer $N > 1$ can be written as $N = \prod_i p_i^{e_i}$, where the $\{p_i\}$ are distinct primes and $e_i \geq 1$ for all i; furthermore, the $\{p_i\}$ and $\{e_i\}$ are uniquely determined up to ordering.

We are familiar with the process of *division with remainder* from elementary school. The following proposition formalizes this notion.

PROPOSITION 7.1 *Let a be an integer and b a positive integer. Then there exist unique integers q, r for which $a = qb + r$ and $0 \leq r < b$.*

Furthermore, given integers a and b as in the proposition, it is possible to compute q and r in polynomial time. See Appendix B.1.

The *greatest common divisor* of two non-negative integers a, b, written $\gcd(a, b)$, is the largest integer c such that $c \mid a$ and $c \mid b$. (We leave $\gcd(0, 0)$ undefined.) The notion of greatest common divisor also makes sense when either or both of a, b are negative but we will never need this; therefore, when we write $\gcd(a, b)$ we always assume that $a, b \geq 0$. Note that $\gcd(b, 0) = \gcd(0, b) = b$; also, if p is prime then $\gcd(a, p)$ is either equal to 1 or p. If $\gcd(a, b) = 1$ we say that a and b are *relatively prime*.

The following is a useful result:

PROPOSITION 7.2 *Let a, b be positive integers. Then there exist integers X, Y such that $Xa + Yb = \gcd(a, b)$. Furthermore, $\gcd(a, b)$ is the smallest positive integer that can be expressed in this way.*

PROOF Consider the set $I \stackrel{\text{def}}{=} \{\hat{X}a + \hat{Y}b \mid \hat{X}, \hat{Y} \in \mathbb{Z}\}$. Note that $a, b \in I$, and so I certainly contains some positive integers. Let d be the smallest positive integer in I. We show that $d = \gcd(a, b)$; since d can be written as $d = Xa + Yb$ for some $X, Y \in \mathbb{Z}$ (because $d \in I$), this proves the theorem.

To show this, we must prove that $d \mid a$ and $d \mid b$, and that d is the largest integer with this property. In fact, we can show that d divides every element in I. To see this, take an arbitrary $c \in I$ and write $c = X'a + Y'b$ with $X', Y' \in \mathbb{Z}$. Using division with remainder (Proposition 7.1) we have that $c = qd + r$ with q, r integers and $0 \leq r < d$. Then

$$r = c - qd = X'a + Y'b - q(Xa + Yb) = (X' - qX)a + (Y' - qY)b \in I.$$

If $r \neq 0$, this contradicts our choice of d as the *smallest* positive integer in I (because $r < d$). So, $r = 0$ and hence $d \mid c$. This shows that d divides every element of I.

Since $a \in I$ and $b \in I$, the above shows that $d \mid a$ and $d \mid b$ and so d is a common divisor of a and b. It remains to show that it is the largest common divisor. Assume there exists an integer $d' > d$ such that $d' \mid a$ and $d' \mid b$. Then by the observation made earlier, $d' \mid Xa + Yb$. Since the latter is equal to d, this means $d' \mid d$. But this is impossible if d' is larger than d. We conclude that d is the largest integer dividing both a and b, and hence $d = \gcd(a, b)$. ■

Given a and b, the *Euclidean algorithm* can be used to compute $\gcd(a, b)$ in polynomial time. The *extended Euclidean algorithm* can be used to compute X, Y (as in the above proposition) in polynomial time as well. See Appendix B.1.2 for details.

The preceding proposition is very useful in proving additional results about divisibility. We show two examples now.

PROPOSITION 7.3 *If $c \mid ab$ and $\gcd(a, c) = 1$, then $c \mid b$. In particular, if p is prime and $p \mid ab$ then either $p \mid a$ or $p \mid b$.*

PROOF Since $c \mid ab$ we can write $\gamma c = ab$ for some integer γ. If $\gcd(a, c) = 1$ then, by the previous proposition, there exist integers X, Y such that $1 = Xa + Yc$. Multiplying both sides by b, we obtain

$$b = Xab + Ycb = X\gamma c + Ycb = c \cdot (X\gamma + Yb).$$

Since $(X\gamma + Yb)$ is an integer, it follows that $c \mid b$.

The second part of the proposition follows from the fact that if $p \nmid a$ then $\gcd(a, p) = 1$. ■

PROPOSITION 7.4 *If $p \mid N$, $q \mid N$, and $\gcd(p, q) = 1$, then $pq \mid N$.*

PROOF Write $pa = N$, $qb = N$, and (using Proposition 7.2) $1 = Xp + Yq$, where a, b, X, Y are all integers. Multiplying both sides of the last equation by N, we obtain

$$N = XpN + YqN = Xpqb + Yqpa = pq(Xb + Ya),$$

showing that $pq \mid N$. ■

7.1.2 Modular Arithmetic

Let $a, b, N \in \mathbb{Z}$ with $N > 1$. We use the notation $[a \bmod N]$ to denote the remainder of a upon division by N. In more detail: by Proposition 7.1 there exist unique q, r with $a = qN + r$ and $0 \leq r < N$, and we define $[a \bmod N]$ to be equal to this r. Note therefore that $0 \leq [a \bmod N] < N$. We refer to the process of mapping a to $[a \bmod N]$ as *reduction modulo N*.

We say that *a and b are congruent modulo N*, written $a = b \bmod N$, if $[a \bmod N] = [b \bmod N]$, i.e., the remainder when a is divided by N is the same as the remainder when b is divided by N. Note that $a = b \bmod N$ if and only if $N \mid (a - b)$. By way of notation, in an expression such as

$$a = b = c = \cdots = z \bmod N,$$

the understanding is that *every* equal sign in this sequence (and not just the last) refers to congruence modulo N.

Note that $a = [b \bmod N]$ implies $a = b \bmod N$, but not vice versa. (On the other hand, $[a \bmod N] = [b \bmod N]$ if and only if $a = b \bmod N$.) For example, $36 = 21 \bmod 15$ but $36 \neq [21 \bmod 15] = 6$.

Congruence modulo N is an equivalence relation: i.e., it is reflexive ($a = a \bmod N$ for all a), symmetric ($a = b \bmod N$ implies $b = a \bmod N$), and transitive (if $a = b \bmod N$ and $b = c \bmod N$ then $a = c \bmod N$). Congruence modulo N also obeys the standard rules of arithmetic with respect to addition, subtraction, and multiplication; so, for example, if $a = a' \bmod N$ and $b = b' \bmod N$ then $(a + b) = (a' + b') \bmod N$ and $ab = a'b' \bmod N$. A consequence is that we can "reduce and then add/multiply" instead of having to "add/multiply and then reduce," a feature which can often be used to simplify calculations.

Example 7.5
Let us compute $[1093028 \cdot 190301 \bmod 100]$. Since $1093028 = 28 \bmod 100$ and $190301 = 1 \bmod 100$, we have

$$\begin{aligned} 1093028 \cdot 190301 &= [1093028 \bmod 100] \cdot [190301 \bmod 100] \bmod 100 \\ &= 28 \cdot 1 = 28 \bmod 100. \end{aligned}$$

The alternative way of calculating the answer (namely, computing the product $1093028 \cdot 190301$ and then reducing the answer modulo 100) is much more time-consuming. ♢

Congruence modulo N does *not* (in general) respect division. That is, if $a = a' \bmod N$ and $b = b' \bmod N$ then it is not necessarily true that $a/b = a'/b' \bmod N$; in fact, the expression "$a/b \bmod N$" is not always well-defined. As a specific example that often causes confusion, $ab = cb \bmod N$ does *not* necessarily imply that $a = c \bmod N$.

Example 7.6
Take $N = 24$. Then $3 \cdot 2 = 6 = 15 \cdot 2 \bmod 24$, but $3 \neq 15 \bmod 24$. ♢

In certain cases, however, we can define a meaningful notion of division. If for a given integer b there exists an integer b^{-1} such that $bb^{-1} = 1 \bmod N$, we say that b^{-1} is a (multiplicative) *inverse* of b modulo N and call b *invertible* modulo N. Clearly, '0' is never invertible. It is also not difficult to show that if β is a multiplicative inverse of b modulo N then so is $[\beta \bmod N]$. Furthermore, if β' is another multiplicative inverse of b then $[\beta \bmod N] = [\beta' \bmod N]$. When b is invertible we can therefore simply let b^{-1} denote the *unique* multiplicative inverse of b that lies in the range $\{1, \ldots, N-1\}$.

When b is invertible modulo N we define division by b modulo N as multiplication by b^{-1} modulo N (i.e., we define $a/b \stackrel{\text{def}}{=} ab^{-1} \bmod N$). We stress that division by b is *only defined* when b is invertible. If $ab = cb \bmod N$ and b is invertible, then we may divide each side of the equation by b (or, equivalently, multiply each side by b^{-1}) to obtain

$$(ab) \cdot b^{-1} = (cb) \cdot b^{-1} \bmod N \quad \Rightarrow \quad a = c \bmod N.$$

We see that in this case, division works "as expected." Invertible integers are therefore "nicer" to work with, in some sense.

The natural question is: which integers are invertible modulo a given modulus N? We can fully answer this question using Proposition 7.2:

PROPOSITION 7.7 *Let a, N be integers, with $N > 1$. Then a is invertible modulo N if and only if* $\gcd(a, N) = 1$.

PROOF Assume a is invertible modulo N, and let b denote its inverse. Note that $a \neq 0$ since $0 \cdot b = 0 \bmod N$ regardless of the value of b. Since $ab = 1 \bmod N$, the definition of congruence modulo N implies that $ab - 1 = cN$ for some $c \in \mathbb{Z}$. Equivalently, $ba - cN = 1$. Since, by Proposition 7.2, $\gcd(a, N)$ is the smallest positive integer that can be expressed in this way, and there is no integer smaller than 1, this implies that $\gcd(a, N) = 1$.

Conversely, if $\gcd(a, N) = 1$ then by Proposition 7.2 there exist integers X, Y such that $Xa + YN = 1$. Reducing each side of this equation modulo N gives $Xa = 1 \bmod N$, and we see that $[X \bmod N]$ is a multiplicative inverse of a. ■

Example 7.8
Let $a = 11$ and $N = 17$. Then $(-3) \cdot 11 + 2 \cdot 17 = 1$, and so $14 = [-3 \bmod 17]$ is the inverse of 11. One can verify that $14 \cdot 11 = 1 \bmod 17$. ♢

Addition, subtraction, multiplication, and computation of inverses (when they exist) modulo N can all be carried out in polynomial time; see Appendix B.2. Exponentiation (i.e., computing $[a^b \bmod N]$ for $b > 0$ an integer) can also be computed in polynomial time; see Appendix B.2.3.

7.1.3 Groups

Let $\mathbb{G}$ be a set. A *binary operation* $\circ$ on $\mathbb{G}$ is simply a function $\circ(\cdot, \cdot)$ that takes as input two elements of $\mathbb{G}$. If $g, h \in \mathbb{G}$ then instead of using the cumbersome notation $\circ(g, h)$, we write $g \circ h$.

We now introduce the important notion of a *group*.

DEFINITION 7.9 *A* group *is a set $\mathbb{G}$ along with a binary operation $\circ$ for which the following conditions hold:*

- (Closure:) *For all $g, h \in \mathbb{G}$, $g \circ h \in \mathbb{G}$.*
- (Existence of an Identity:) *There exists an* identity *$e \in \mathbb{G}$ such that for all $g \in \mathbb{G}$, $e \circ g = g = g \circ e$.*
- (Existence of Inverses:) *For all $g \in \mathbb{G}$ there exists an element $h \in \mathbb{G}$ such that $g \circ h = e = h \circ g$. Such an h is called an* inverse *of g.*
- (Associativity:) *For all $g_1, g_2, g_3 \in \mathbb{G}$, $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$.*

When $\mathbb{G}$ has a finite number of elements, we say $\mathbb{G}$ is a finite group *and let $|\mathbb{G}|$ denote the* order *of the group; that is, the number of elements in $\mathbb{G}$.*

A group $\mathbb{G}$ with operation $\circ$ is abelian *if the following holds:*

- (Commutativity:) *For all $g, h \in \mathbb{G}$, $g \circ h = h \circ g$.*

When the binary operation is understood, we simply call the set $\mathbb{G}$ a group.

We will always deal with finite, abelian groups. We will be careful to specify, however, when a result requires these assumptions.

Associativity implies that we do not need to include parentheses when writing long expressions; that is, the notation $g_1 \circ g_2 \circ \cdots \circ g_n$ is unambiguous since it does not matter in what order we evaluate the operation $\circ$.

One can show that the identity element in a group $\mathbb{G}$ is *unique*, and so we can therefore refer to *the* identity of a group. One can also show that each element g of a group has a *unique* inverse. See Exercise 7.1.

If $\mathbb{G}$ is a group, a set $\mathbb{H} \subseteq \mathbb{G}$ is a *subgroup of* $\mathbb{G}$ if $\mathbb{H}$ itself forms a group under the same operation associated with $\mathbb{G}$. To check that $\mathbb{H}$ is a subgroup, we need to verify closure, existence of identity and inverses, and associativity as per Definition 7.9. (Actually, associativity — as well as commutativity if $\mathbb{G}$ is abelian — is inherited automatically from $\mathbb{G}$.) Every group $\mathbb{G}$ always has the trivial subgroups $\mathbb{G}$ and $\{1\}$. We call $\mathbb{H}$ a *strict* subgroup of $\mathbb{G}$ if $\mathbb{H} \neq \mathbb{G}$.

In general, we will not use the notation $\circ$ to denote the group operation. Instead, we will use either *additive* notation or *multiplicative* notation depending on the group under discussion. When using additive notation, the group operation applied to two elements g, h is denoted $g + h$; the identity is denoted by '0', and the inverse of an element g is denoted by $-g$. When using multiplicative notation, the group operation applied to g, h is denoted by $g \cdot h$ or simply gh; the identity is denoted by '1', and the inverse of an element g is denoted by g^{-1}. As in the case of multiplication modulo N, we also define division by g as multiplication by g^{-1} (i.e., h/g is defined to mean hg^{-1}). When we state general results, we will always use multiplicative notation. *This does not imply that the group operation corresponds to integer addition or multiplication. This merely serves as useful notation.*

At this point, it may be helpful to see some examples.

Example 7.10

A set may be a group under one operation, but not another. For example, the set of integers $\mathbb{Z}$ is an abelian group under addition: the identity is the element '0', and every integer g has inverse $-g$. On the other hand, it is not a group under multiplication since, for example, the integer '2' does not have a multiplicative inverse in the integers. ♢

Example 7.11

The set of real numbers $\mathbb{R}$ is not a group under multiplication, since '0' does not have a multiplicative inverse. The set of *non-zero* real numbers, however, is an abelian group under multiplication with identity '1'. ♢

The following example introduces the group $\mathbb{Z}_N$ that we will use frequently.

Example 7.12

Let $N \geq 2$ be an integer. The set $\{0, \ldots, N-1\}$ with respect to addition modulo N (i.e., where $a + b \stackrel{\text{def}}{=} [a + b \bmod N]$) is an abelian group of order N: Closure is obvious; associativity and commutativity follow from the fact that the integers satisfy these properties; the identity is 0; and, since $a + (N - a) = 0 \bmod N$, it follows that the inverse of any element a is $[(N - a) \bmod N]$. We denote this group by $\mathbb{Z}_N$. (We will also use $\mathbb{Z}_N$ to denote the set $\{0, \ldots, N-1\}$ without regard to any particular group operation.) ♢

We end this section with an easy lemma that formalizes an obvious "cancelation law" for groups.

LEMMA 7.13 *Let $\mathbb{G}$ be a group and $a, b, c \in \mathbb{G}$. If $ac = bc$, then $a = b$. In particular, if $ac = c$ then a is the identity in $\mathbb{G}$.*

PROOF We know $ac = bc$. Multiplying both sides by the unique inverse c^{-1} of c, we obtain $a = b$. In detail:

$$ac = bc \;\Rightarrow\; (ac)c^{-1} = (bc)\cdot c^{-1} \;\Rightarrow\; a(cc^{-1}) = b(cc^{-1}) \;\Rightarrow\; a \cdot 1 = b \cdot 1 \;\Rightarrow a = b.$$

■

Compare the above proof to the discussion (preceding Proposition 7.7) regarding a cancelation law for division modulo N. As indicated by the similarity, the *invertible* elements modulo N form a group under multiplication modulo N. We will return to this example in more detail shortly.

Group Exponentiation

It is often useful to be able to describe the group operation applied m times to a fixed element g, where m is a positive integer. When using additive notation, we express this as $m \cdot g$ or mg; that is,

$$mg = m \cdot g \stackrel{\text{def}}{=} \underbrace{g + \cdots + g}_{m \text{ times}}.$$

Note that m is an *integer*, while g is a *group element*. So mg does *not* represent the group operation applied to m and g (indeed, we are working in a group where the group operation is written additively). Thankfully, however, the notation "behaves as it should"; so, for example, if $g \in \mathbb{G}$ and m, m' are integers then $(mg) + (m'g) = (m + m')g$, $m(m'g) = (mm')g$, and $1 \cdot g = g$. In an *abelian* group $\mathbb{G}$ with $g, h \in \mathbb{G}$, $(mg) + (mh) = m(g + h)$.

When using multiplicative notation, we express application of the group operation m times to an element g by g^m. That is,

$$g^m \stackrel{\text{def}}{=} \underbrace{g \cdots g}_{m \text{ times}}.$$

The familiar rules of exponentiation hold: $g^m \cdot g^{m'} = g^{m+m'}$, $(g^m)^{m'} = g^{mm'}$, and $g^1 = g$. Also, if $\mathbb{G}$ is an abelian group and $g, h \in \mathbb{G}$ then $g^m \cdot h^m = (gh)^m$. Note that all these results are simply "translations" of the results stated in

the previous paragraph to the setting of groups written multiplicatively rather than additively.

The above notation is extended in the natural way to the case when m is zero or a negative integer. (In general, we leave g^m undefined if m is not an integer.) When using additive notation we have $0 \cdot g \stackrel{\text{def}}{=} 0$ and $(-m) \cdot g \stackrel{\text{def}}{=} m \cdot (-g)$ for m a positive integer. (Note that in the equation '$0 \cdot g = 0$' the '0' on the left-hand side is the integer 0 while the '0' on the right-hand side is the identity element in the group.) As one would expect, it can be shown that $(-m) \cdot g = -(mg)$. When using multiplicative notation, $g^0 \stackrel{\text{def}}{=} 1$ and $g^{-m} \stackrel{\text{def}}{=} (g^{-1})^m$. Again, as expected, one can show that $g^{-m} = (g^m)^{-1}$.

Let $g \in \mathbb{G}$ and $b \geq 0$ be an integer. Then the exponentiation g^b can be computed using a polynomial number of underlying group operations in $\mathbb{G}$. Thus, if the group operation can be computed in polynomial time then so can exponentiation. This is discussed in Appendix B.2.3.

We now know enough to prove the following remarkable result:

THEOREM 7.14 *Let $\mathbb{G}$ be a finite group with $m = |\mathbb{G}|$, the order of the group. Then for any element $g \in \mathbb{G}$, $g^m = 1$.*

PROOF We prove the theorem only when $\mathbb{G}$ is abelian (though it holds for any finite group). Fix arbitrary $g \in \mathbb{G}$, and let $g_1, \ldots, g_m$ be the elements of $\mathbb{G}$. We claim that

$$g_1 \cdot g_2 \cdots g_m = (gg_1) \cdot (gg_2) \cdots (gg_m).$$

To see this, note that $gg_i = gg_j$ implies $g_i = g_j$ by Lemma 7.13. So each of the m elements in parentheses on the right-hand side of the displayed equation is distinct. Because there are exactly m elements in $\mathbb{G}$, the m elements being multiplied together on the right-hand side are simply all elements of $\mathbb{G}$ in some permuted order. Since $\mathbb{G}$ is abelian the order in which all elements of the group are multiplied does not matter, and so the right-hand side is equal to the left-hand side.

Again using the fact that $\mathbb{G}$ is abelian, we can "pull out" all occurrences of g and obtain

$$g_1 \cdot g_2 \cdots g_m = (gg_1) \cdot (gg_2) \cdots (gg_m) = g^m \cdot (g_1 \cdot g_2 \cdots g_m).$$

Appealing once again to Lemma 7.13, this implies $g^m = 1$. ■

An important corollary of the above is that we can work "modulo the group order in the exponent":

COROLLARY 7.15 *Let $\mathbb{G}$ be a finite group with $m = |\mathbb{G}| > 1$. Then for any $g \in \mathbb{G}$ and any integer i, we have $g^i = g^{[i \bmod m]}$.*

PROOF Say $i = qm + r$, where q, r are integers and $r = [i \bmod m]$. Using Theorem 7.14,

$$g^i = g^{qm+r} = g^{qm} \cdot g^r = (g^m)^q \cdot g^r = 1^q \cdot g^r = g^r,$$

as claimed. ■

Example 7.16
Written additively, the above corollary says that if g is an element in a group of order m, then $i \cdot g = [i \bmod m] \cdot g$. As an example, consider the group $\mathbb{Z}_{15}$ of order $m = 15$, and take $g = 11$. The corollary says that

$$152 \cdot 11 = [152 \bmod 15] \cdot 11 = 2 \cdot 11 = 11 + 11 = 22 = 7 \bmod 15.$$

The above exactly agrees with the fact (cf. Example 7.5) that we can "reduce and then multiply" rather than having to "multiply and then reduce." ♢

Another corollary that will be extremely useful for cryptographic applications is the following:

COROLLARY 7.17 *Let $\mathbb{G}$ be a finite group with $m = |\mathbb{G}| > 1$. Let $e > 0$ be an integer, and define the function $f_e : \mathbb{G} \to \mathbb{G}$ by $f_e(g) = g^e$. If $\gcd(e, m) = 1$, then f_e is a permutation (i.e., a bijection). Moreover, if $d = [e^{-1} \bmod m]$ then f_d is the inverse of f_e.*

PROOF By Proposition 7.7, $\gcd(e, m) = 1$ implies that e is invertible modulo m and, in this case, d is the multiplicative inverse of e modulo m. The second part of the claim implies the first, so we need only show that f_d is the inverse of f_e. This is true because for any $g \in \mathbb{G}$ we have

$$f_d\left(f_e(g)\right) = f_d(g^e) = (g^e)^d = g^{ed} = g^{[ed \bmod m]} = g^1 = g,$$

where the fourth equality follows from Corollary 7.15. ■

7.1.4 The Group $\mathbb{Z}_N^*$

As discussed in Example 7.12, the set $\mathbb{Z}_N = \{0, \ldots, N-1\}$ is a group under addition modulo N. Can we define a group structure over the set $\{0, \ldots, N-1\}$ with respect to *multiplication* modulo N? In doing so, we will have to eliminate those elements in this set that are not invertible; for example, we will have to eliminate '0' since it obviously has no multiplicative inverse. This is not the only potential problem: if $N = 6$, then '3' is not invertible as can be proved by exhaustively trying every possibility.

Which elements $a \in \{1, \ldots, N-1\}$ are invertible modulo N? Proposition 7.7 says that these are exactly those elements a for which $\gcd(a, N) = 1$. We have also seen in Section 7.1.2 that whenever a is invertible, it has an inverse lying in the range $\{1, \ldots, N-1\}$. This leads us to define, for $N > 1$, the set

$$\mathbb{Z}_N^* \stackrel{\text{def}}{=} \{a \in \{1, \ldots, N-1\} \mid \gcd(a, N) = 1\};$$

i.e., $\mathbb{Z}_N^*$ consists of integers in the set $\{1, \ldots, N-1\}$ that are relatively prime to N. The group operation is multiplication modulo N; i.e., $ab \stackrel{\text{def}}{=} [ab \bmod N]$.

We claim that $\mathbb{Z}_N^*$ is an abelian group with respect to this operation. Since the element '1' is always in $\mathbb{Z}_N^*$, the set clearly contains an identity element. The discussion above shows that each element in $\mathbb{Z}_N^*$ has a multiplicative inverse in the same set. Commutativity and associativity follow from the fact that these properties hold over the integers. To show that closure holds, let $a, b \in \mathbb{Z}_N^*$, let $c = [ab \bmod N]$, and assume $c \notin \mathbb{Z}_N^*$. This means that $\gcd(c, N) \neq 1$, and so there exists a prime p dividing both N and c. Since $ab = qN + c$ for some integer q, we see that $p \mid ab$. By Proposition 7.3, this means $p \mid a$ or $p \mid b$; but then either $\gcd(a, N) \neq 1$ or $\gcd(b, N) \neq 1$, contradicting our assumption that $a, b \in \mathbb{Z}_N^*$.

Summarizing:

PROPOSITION 7.18 *Let $N > 1$ be an integer. Then $\mathbb{Z}_N^*$ is an abelian group under multiplication modulo N.*

Define $\phi(N) \stackrel{\text{def}}{=} |\mathbb{Z}_N^*|$, the order of the group $\mathbb{Z}_N^*$ (ϕ is called the *Euler phi function*). What is the value of $\phi(N)$? First consider the case when $N = p$ is prime. Then *all* elements in $\{1, \ldots, p-1\}$ are relatively prime to p, and so $\phi(p) = |\mathbb{Z}_p^*| = p - 1$. Next consider the case that $N = pq$, where p, q are distinct primes. If an integer $a \in \{1, \ldots, N-1\}$ is not relatively prime to N, then either $p \mid a$ or $q \mid a$ (a cannot be divisible by both p and q since this would imply $pq \mid a$ but $a < N = pq$). The elements in $\{1, \ldots, N-1\}$ divisible by p are exactly the $(q-1)$ elements $p, 2p, 3p, \ldots, (q-1)p$, and the elements divisible by q are exactly the $(p-1)$ elements $q, 2q, \ldots, (p-1)q$. The number of elements remaining (i.e., those that are neither divisible by p or q) is therefore given by

$$(N-1) - (q-1) - (p-1) = pq - p - q + 1 = (p-1)(q-1).$$

We have thus proved that $\phi(N) = (p-1)(q-1)$ when N is the product of two distinct primes p and q.

You are asked to prove the following general result (used only rarely in the rest of the book) in Exercise 7.4:

THEOREM 7.19 *Let $N = \prod_i p_i^{e_i}$, where the $\{p_i\}$ are distinct primes and $e_i \geq 1$. Then $\phi(N) = \prod_i p_i^{e_i - 1}(p_i - 1)$.*

Example 7.20
Take $N = 15 = 5 \cdot 3$. Then $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ and $|\mathbb{Z}_{15}^*| = 8 = 4 \cdot 2 = \phi(15)$. The inverse of 8 in $\mathbb{Z}_{15}^*$ is 2, since $8 \cdot 2 = 16 = 1 \bmod 15$. $\diamondsuit$

We have shown that $\mathbb{Z}_N^*$ is a group of order $\phi(N)$. The following are now easy corollaries of Theorem 7.14 and Corollary 7.17:

COROLLARY 7.21 *Take arbitrary $N > 1$ and $a \in \mathbb{Z}_N^*$. Then*

$$a^{\phi(N)} = 1 \bmod N.$$

For the specific case that $N = p$ is prime and $a \in \{1, \ldots, p-1\}$, we have

$$a^{p-1} = 1 \bmod p.$$

COROLLARY 7.22 *Fix $N > 1$. For integer $e > 0$ define $f_e : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$ by $f_e(x) = [x^e \bmod N]$. If e is relatively prime to $\phi(N)$ then f_e is a permutation. Moreover, if $d = [e^{-1} \bmod \phi(N)]$ then f_d is the inverse of f_e.*

7.1.5 * Isomorphisms and the Chinese Remainder Theorem

An *isomorphism* of a group $\mathbb{G}$ provides an alternative, but equivalent, way of thinking about $\mathbb{G}$.

DEFINITION 7.23 *Let $\mathbb{G}, \mathbb{H}$ be groups with respect to the operations $\circ_{\mathbb{G}}, \circ_{\mathbb{H}}$, respectively. A function $f : \mathbb{G} \to \mathbb{H}$ is an* isomorphism from $\mathbb{G}$ to $\mathbb{H}$ *if:*

1. *f is a bijection, and*
2. *For all $g_1, g_2 \in \mathbb{G}$ we have $f(g_1 \circ_{\mathbb{G}} g_2) = f(g_1) \circ_{\mathbb{H}} f(g_2)$.*

If there exists an isomorphism from $\mathbb{G}$ to $\mathbb{H}$ then we say that these groups are isomorphic *and write this as $\mathbb{G} \simeq \mathbb{H}$.*

In essence, an isomorphism from $\mathbb{G}$ to $\mathbb{H}$ is just a *renaming* of elements of $\mathbb{G}$ as elements of $\mathbb{H}$. Note that if $\mathbb{G}$ is finite and $\mathbb{G} \simeq \mathbb{H}$, then $\mathbb{H}$ must be finite and of the same size as $\mathbb{G}$. Also, if there exists an isomorphism f from $\mathbb{G}$ to $\mathbb{H}$ then f^{-1} is an isomorphism from $\mathbb{H}$ to $\mathbb{G}$. However, it is possible that f may be efficiently computable while f^{-1} is not (or vice versa).

The aim of this section is to use the language of isomorphisms to better understand the group structure of $\mathbb{Z}_N$ and $\mathbb{Z}_N^*$ when $N = pq$ is a product of two distinct primes. We first need to introduce the notion of a *cross product* of groups. Given groups $\mathbb{G}, \mathbb{H}$ with group operations $\circ_{\mathbb{G}}, \circ_{\mathbb{H}}$ respectively, we define a new group $\mathbb{G} \times \mathbb{H}$ (the *cross product of* $\mathbb{G}$ *and* $\mathbb{H}$) as follows. The

elements of $\mathbb{G} \times \mathbb{H}$ are ordered pairs (g, h) with $g \in \mathbb{G}$ and $h \in \mathbb{H}$; thus, if $\mathbb{G}$ has n elements and $\mathbb{H}$ has n' elements, $\mathbb{G} \times \mathbb{H}$ has $n \cdot n'$ elements. The group operation $\circ$ on $\mathbb{G} \times \mathbb{H}$ is applied component-wise; that is:

$$(g, h) \circ (g', h') \stackrel{\text{def}}{=} (g \circ_{\mathbb{G}} g',\ h \circ_{\mathbb{H}} h').$$

We leave it to Exercise 7.7 to verify that $\mathbb{G} \times \mathbb{H}$ is indeed a group. The above notation can be extended to cross products of more than two groups in the natural way, though we will not need this for what follows.

We may now state and prove the *Chinese remainder theorem.*

THEOREM 7.24 (Chinese Remainder Theorem) *Let $N = pq$ where p and q are relatively prime. Then*

$$\mathbb{Z}_N \simeq \mathbb{Z}_p \times \mathbb{Z}_q \quad \textit{and} \quad \mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*.$$

Moreover, let f be the function mapping elements $x \in \{0, \ldots, N-1\}$ to pairs (x_p, x_q) with $x_p \in \{0, \ldots, p-1\}$ and $x_q \in \{0, \ldots, q-1\}$ defined by

$$f(x) \stackrel{\text{def}}{=} ([x \bmod p], [x \bmod q]).$$

Then f is an isomorphism from $\mathbb{Z}_N$ to $\mathbb{Z}_p \times \mathbb{Z}_q$ as well as an isomorphism from $\mathbb{Z}_N^*$ to $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.

PROOF It is clear that for any $x \in \mathbb{Z}_N$ the output $f(x)$ is a pair of elements (x_p, x_q) with $x_p \in \mathbb{Z}_p$ and $x_q \in \mathbb{Z}_q$. Furthermore, we claim that if $x \in \mathbb{Z}_N^*$ then $(x_p, x_q) \in \mathbb{Z}_p^* \times \mathbb{Z}_q^*$. Indeed, if $x_p \notin \mathbb{Z}_p^*$ then this means that $\gcd([x \bmod p], p) \neq 1$. But then $\gcd(x, p) \neq 1$. This implies $\gcd(x, N) \neq 1$, contradicting the assumption that $x \in \mathbb{Z}_N^*$. (An analogous argument holds if $x_q \notin \mathbb{Z}_q^*$.)

We now show that f is an isomorphism from $\mathbb{Z}_N$ to $\mathbb{Z}_p \times \mathbb{Z}_q$. (The proof that it is an isomorphism from $\mathbb{Z}_N^*$ to $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$ is similar.) Let us start by proving that f is one-to-one. Say $f(x) = (x_p, x_q) = f(x')$. Then $x = x_p = x' \bmod p$ and $x = x_q = x' \bmod q$. This in turn implies that $(x - x')$ is divisible by both p and q. Since $\gcd(p, q) = 1$, Proposition 7.4 says that $pq = N$ divides $(x - x')$. But then $x = x' \bmod N$. For $x, x' \in \mathbb{Z}_N$, this means that $x = x'$ and so f is indeed one-to-one. Since $|\mathbb{Z}_N| = N = p \cdot q = |\mathbb{Z}_p| \cdot |\mathbb{Z}_q|$, the sizes of $\mathbb{Z}_N$ and $\mathbb{Z}_p \times \mathbb{Z}_q$ are the same. This in combination with the fact that f is one-to-one implies that f is bijective.

In the following paragraph, let $+_N$ denote addition modulo N, and let $\boxplus$ denote the group operation in $\mathbb{Z}_p \times \mathbb{Z}_q$ (i.e., addition modulo p in the first component and addition modulo q in the second component). To conclude the proof that f is an isomorphism from $\mathbb{Z}_N$ to $\mathbb{Z}_p \times \mathbb{Z}_q$, we need to show that for all $a, b \in \mathbb{Z}_N$ it holds that $f(a +_N b) = f(a) \boxplus f(b)$.

To see that this is true, note that

$$\begin{aligned} f(a +_N b) &= \Big([(a +_N b) \bmod p],\, [(a +_N b) \bmod q]\Big) \\ &= \Big([(a + b) \bmod p],\, [(a + b) \bmod q]\Big) \\ &= \Big([a \bmod p],\, [a \bmod q]\Big) \boxplus \Big([b \bmod p],\, [b \bmod q]\Big) = f(a) \boxplus f(b). \end{aligned}$$

(For the second equality, above, we use the fact that $[[X \bmod N] \bmod p\,] = [[X \bmod p\,] \bmod p\,]$ when $p \,|\, N$; see Exercise 7.8.) ■

The theorem does not require p or q to be prime. An extension of the Chinese remainder theorem says that if $p_1, p_2, \ldots, p_\ell$ are pairwise relatively prime (i.e., $\gcd(p_i, p_j) = 1$ for all $i \neq j$) and $N \stackrel{\text{def}}{=} \prod_{i=1}^{\ell} p_i$, then

$$\mathbb{Z}_N \simeq \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_\ell} \quad \text{and} \quad \mathbb{Z}_N^* \simeq \mathbb{Z}_{p_1}^* \times \cdots \times \mathbb{Z}_{p_\ell}^*.$$

An isomorphism in each case is obtained by a natural extension of the one used in the theorem above.

By way of notation, with N understood and $x \in \{0, 1, \ldots, N-1\}$ we write $x \leftrightarrow (x_p, x_q)$ for $x_p = [x \bmod p]$ and $x_q = [x \bmod q]$. I.e., $x \leftrightarrow (x_p, x_q)$ if and only if $f(x) = (x_p, x_q)$, where f is as in the theorem above. One way to think about this notation is that it means "x (in $\mathbb{Z}_N$) *corresponds to* (x_p, x_q) (in $\mathbb{Z}_p \times \mathbb{Z}_q$)." The same notation is used when dealing with $x \in \mathbb{Z}_N^*$.

Example 7.25
Take $15 = 5 \cdot 3$, and consider $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$. The Chinese remainder theorem says that this group is isomorphic to $\mathbb{Z}_5^* \times \mathbb{Z}_3^*$. Indeed, we can compute

$$\begin{array}{l} 1 \leftrightarrow (1,1)\ \ 2 \leftrightarrow (2,2)\ \ \ 4 \leftrightarrow (4,1)\ \ \ 7 \leftrightarrow (2,1) \\ 8 \leftrightarrow (3,2)\ 11 \leftrightarrow (1,2)\ 13 \leftrightarrow (3,1)\ 14 \leftrightarrow (4,2)\,' \end{array}$$

where each possible pair (a, b) with $a \in \mathbb{Z}_5^*$ and $b \in \mathbb{Z}_3^*$ appears exactly once. ♢

Using the Chinese Remainder Theorem

If two groups are isomorphic, then they both serve as representations of the same underlying "algebraic structure." Nevertheless, the choice of which representation to use can affect the *computational efficiency* of group operations. We show this abstractly, and then in the specific context of $\mathbb{Z}_N$ and $\mathbb{Z}_N^*$.

Let $\mathbb{G}, \mathbb{H}$ be groups with operations $\circ_{\mathbb{G}}, \circ_{\mathbb{H}}$, respectively, and say f is an isomorphism from $\mathbb{G}$ to $\mathbb{H}$ where both f and f^{-1} can be computed efficiently (in general this need not be the case). Then for $g_1, g_2 \in \mathbb{G}$ we can compute

$g = g_1 \circ_{\mathbb{G}} g_2$ in two ways: either by directly computing the group operation in $\mathbb{G}$, or by carrying out the following steps:

1. Compute $h_1 = f(g_1)$ and $h_2 = f(g_2)$;
2. Compute $h = h_1 \circ_{\mathbb{H}} h_2$ using the group operation in $\mathbb{H}$;
3. Compute $g = f^{-1}(h)$.

Which method is better depends on the specific groups under consideration, as well as the efficiency of computing f and f^{-1}.

We now turn to the specific case of computations modulo N, when $N = pq$ is a product of distinct primes. The Chinese remainder theorem shows that addition or multiplication modulo N can be "transformed" to analogous operations modulo p and q. (Moreover, an easy corollary of the Chinese remainder theorem shows that this holds true for exponentiation as well.) Using Exercise 7.25, we can show some simple examples with $N = 15$.

Example 7.26
Say we wish to compute the product $14 \cdot 13$ modulo 15 (i.e., in $\mathbb{Z}_{15}^*$). Exercise 7.25 gives $14 \leftrightarrow (4, 2)$ and $13 \leftrightarrow (3, 1)$. Now,

$$[14 \cdot 13 \bmod 15] \leftrightarrow (4, 2) \cdot (3, 1) = ([4 \cdot 3 \bmod 5],\ [2 \cdot 1 \bmod 3]) = (2, 2).$$

But $(2, 2) \leftrightarrow 2$, which is the correct answer since $14 \cdot 13 = 2 \bmod 15$. ♢

Example 7.27
Say we wish to compute $11^2 \bmod 15$. Example 7.25 gives $11 \leftrightarrow (1, 2)$ and so $[11^2 \bmod 15] \leftrightarrow (1, 2)^2$, where on the right-hand side exponentiation is in the group $\mathbb{Z}_5^* \times \mathbb{Z}_3^*$. Thus,

$$[11^2 \bmod 15] \leftrightarrow (1, 2)^2 = (1^2 \bmod 5,\ 2^2 \bmod 3) = (1, 1) \leftrightarrow 1.$$

Indeed, $11^2 = 1 \bmod 15$. ♢

One thing we have not yet discussed is how to algorithmically convert back-and-forth between the representation of an element modulo N and its representation modulo p and q. We now show that the conversion can be carried out in polynomial time *provided the factorization of N is known.*

It is easy to map an element x modulo N to its corresponding representation modulo p and q: the element x corresponds to $([x \bmod p], [x \bmod q])$. Since both the necessary modular reductions can be carried out efficiently (cf. Appendix B.2), this process can be carried out in polynomial time.

For the other direction, we make use of the following observation: an element with representation (x_p, x_q) can be written as

$$(x_p, x_q) = x_p \cdot (1, 0) + x_q \cdot (0, 1).$$

So, if we can find elements $1_p, 1_q \in \{0, \ldots, N-1\}$ such that $1_p \leftrightarrow (1,0)$ and $1_q \leftrightarrow (0,1)$, then (appealing to the Chinese remainder theorem) we know that

$$(x_p,\, x_q) \leftrightarrow [(x_p \cdot 1_p + x_q \cdot 1_q) \bmod N].$$

Since p, q are distinct primes, $\gcd(p,q) = 1$. We can use the extended Euclidean algorithm (cf. Appendix B.1.2) to find integers X, Y such that

$$Xp + Yq = 1.$$

We claim that $1_p = [Yq \bmod N]$. This is because

$$\Big[[Yq \bmod N] \bmod p\Big] = [Yq \bmod p] = [(1 - Xp) \bmod p] = 1$$

and

$$[Yq \bmod N] \bmod q = [Yq \bmod q] = 0\,;$$

and so $Yq \bmod N \leftrightarrow (1,0)$ as desired. In a similar way it can be shown that $1_q = [Xp \bmod N]$.

In summary, we can convert an element represented as (x_p, x_q) to its representation modulo N in the following way (assuming p and q are known):

1. Compute X, Y such that $Xp + Yq = 1$.
2. Set $1_p = [Yq \bmod N]$ and $1_q = [Xp \bmod N]$.
3. Compute $x = [(x_p \cdot 1_p + x_q \cdot 1_q) \bmod N]$.

Note that if many such conversions will be performed, then $1_p, 1_q$ can be computed once-and-for-all in a preprocessing phase.

Example 7.28

Take $p = 5$, $q = 7$, and $N = 5 \cdot 7 = 35$. Say we are given the representation $(4,3)$ and want to convert this to the corresponding element of $\mathbb{Z}_{35}$. Using the extended Euclidean algorithm, we compute

$$3 \cdot 5 - 2 \cdot 7 = 1.$$

Thus, $1_p = [(-2 \cdot 7) \bmod 35] = 21$ and $1_q = [3 \cdot 5 \bmod 35] = 15$. So

$$\begin{aligned}(4,3) &= 4 \cdot (1,0) + 3 \cdot (0,1) \\ &\leftrightarrow [4 \cdot 1_p + 3 \cdot 1_q \bmod 35] \\ &= [4 \cdot 21 + 3 \cdot 15 \bmod 35] \;=\; 24.\end{aligned}$$

Since $24 = 4 \bmod 5$ and $24 = 3 \bmod 7$, this is indeed correct. ♢

Example 7.29
Say we want to compute $[29^{100} \bmod 35]$. We first compute the correspondence $29 \leftrightarrow ([29 \bmod 5], \quad [29 \bmod 7]) = (-1, 1)$. Using the Chinese remainder theorem, we have

$$[29^{100} \bmod 35] \leftrightarrow (1, -1)^{100} = (1^{100} \bmod 5, (-1)^{100} \bmod 7) = (1, 1),$$

and it is immediate that $(1, 1) \leftrightarrow 1$. We conclude that $1 = [29^{100} \bmod 35]$. $\diamondsuit$

Example 7.30
Say we want to compute $[18^{25} \bmod 35]$. We have $18 \leftrightarrow (3, 4)$ and so

$$18^{25} \bmod 35 \leftrightarrow (3, 4)^{25} = ([3^{25} \bmod 5], [4^{25} \bmod 7]).$$

Since $\mathbb{Z}_5^*$ is a group of order 4, we can "work modulo 4 in the exponent" (cf. Corollary 7.15) and see that

$$3^{25} = 3^{25 \bmod 4} = 3^1 = 3 \bmod 5.$$

Similarly,

$$4^{25} = 4^{25 \bmod 6} = 4^1 = 4 \bmod 7.$$

Thus, $([3^{25} \bmod 5], [4^{25} \bmod 7]) = (3, 4) \leftrightarrow 18$ and so $[18^{25} \bmod 35] = 18$. $\diamondsuit$

7.2 Primes, Factoring, and RSA

In this section, we show the first examples of number-theoretic problems that are conjectured to be "hard". We begin with a discussion of one of the oldest problems: *integer factorization* or just *factoring*.

Given a composite integer N, the factoring problem is to find positive integers p, q such that $pq = N$. Factoring is a classic example of a hard problem, both because it is so simple to describe and also because it has been recognized as a hard computational problem for a long time (even before its use in cryptography). The problem can be solved in *exponential* time $\mathcal{O}(\sqrt{N} \cdot \mathsf{polylog}(N))$ using *trial division*: that is, by exhaustively checking whether p divides N for $p = 2, \ldots, \lfloor \sqrt{N} \rfloor$. (This method requires $\sqrt{N}$ divisions, each one taking $\mathsf{polylog}(N) = (\log N)^c$ time for some constant c.) This always succeeds because although the *largest* prime factor of N may be as large as $N/2$, the *smallest* prime factor of N can be at most $\lfloor \sqrt{N} \rfloor$. Although algorithms with better running time are known (see Chapter 8), no *polynomial-time* algorithm that solves the factoring problem has been developed, despite many years of effort.

Consider the following experiment for a given algorithm $\mathcal{A}$ and parameter n:

The weak factoring experiment $\mathsf{w\text{-}Factor}_{\mathcal{A}}(n)$**:**

1. *Choose two n-bit integers x_1, x_2 at random.*
2. *Compute $N := x_1 \cdot x_2$.*
3. *$\mathcal{A}$ is given N, and outputs x'_1, x'_2.*
4. *The output of the experiment is defined to be 1 if $x'_1 \cdot x'_2 = N$, and 0 otherwise.*

We have just said that the factoring problem is believed to be hard. Does this mean that for any PPT algorithm $\mathcal{A}$ we have

$$\Pr[\mathsf{w\text{-}Factor}_{\mathcal{A}}(n) = 1] \leq \mathsf{negl}(n)$$

for some negligible function negl? Not at all. For starters, the number N in the above experiment is *even* with probability 3/4 (as this occurs when either x_1 or x_2 is even) and it is, of course, easy for $\mathcal{A}$ to factor N in this case. While we can make $\mathcal{A}$'s job more difficult by requiring $\mathcal{A}$ to output integers x'_1, x'_2 of length n (as suggested in Chapter 6), it remains the case that x_1 or x_2 (and hence N) might have small prime factors that can still be easily found by $\mathcal{A}$. In cryptographic contexts, we would like to prevent this.

As this discussion indicates, the "hardest" numbers to factor seem to be those having only large prime factors. This suggests re-defining the above experiment so that x_1, x_2 are random n-bit *primes* rather than random n-bit *integers*, and in fact such an experiment will be used when we formally define the factoring assumption in Section 7.2.3. For this experiment to be useful in a cryptographic setting, however, it will be necessary to be able to generate random n-bit primes *efficiently*. This is the topic of the next section.

7.2.1 Generating Random Primes

The same general approach discussed in Appendix B.2.4 for choosing random integers in a certain range can be used to generate random n-bit primes. (The discussion in Appendix B.2.4 is helpful, but not essential, for what follows.) Specifically, we can generate a random n-bit prime by repeatedly choosing random n-bit integers until we find the first prime; we repeat this at most t times. See Algorithm 7.31 for a high-level description of the process.

Note that the algorithm forces the output to be an integer of length *exactly* n (rather than length *at most* n) by fixing the high-order bit of p to '1'. Our convention throughout this book is that an "integer of length n" means an integer whose binary representation *with most significant bit equal to 1* is exactly n bits long.

Given a method that always correctly determines whether or not a given integer p is prime, the above algorithm outputs a *random* n-bit prime conditioned on the event that it does not output fail. The probability that the

ALGORITHM 7.31
Generating a random prime — high-level outline

Input: Length n; parameter t
Output: A random n-bit prime

for $i = 1$ to t: {
  $p' \leftarrow \{0,1\}^{n-1}$
  $p := 1 \| p'$
  if p is prime **return** p
  }
return fail

algorithm outputs fail depends on t, and for our purposes we will want to set t so as to obtain a failure probability that is negligible in n. To show that this approach leads to an *efficient* (i.e., polynomial-time in n) algorithm for generating primes, we need a better understanding of two issues: (1) the probability that a randomly-selected n-bit integer is prime; and (2) how to efficiently test whether a given integer p is prime. We discuss these issues briefly now, and defer a more in-depth exploration of the second topic to Section 7.2.2.

The distribution of primes. The *prime number theorem*, an important result in mathematics, gives fairly precise bounds on the fraction of integers of a given length that are prime. For our purposes, we need only the following weak version of that result:

THEOREM 7.32 *There exists a constant c such that, for any $n > 1$, the number of n-bit primes is at least $c \cdot 2^{n-1}/n$.*

We do not give a proof of this theorem here, though somewhat elementary proofs are known (see the references at the end of the chapter). The theorem implies that the probability that a random n-bit integer is prime is at least

$$\frac{c \cdot 2^{n-1}/n}{2^{n-1}} = \frac{c}{n}.$$

Returning to the approach for generating primes described above, this implies that if we set $t = n^2/c$ then the probability that a prime is *not* chosen in all t iterations of the algorithm is at most

$$\left(1 - \frac{c}{n}\right)^t = \left(\left(1 - \frac{c}{n}\right)^{n/c}\right)^n \leq \left(e^{-1}\right)^n = e^{-n}$$

(using Inequality A.2), which is negligible in n. Thus, using $\mathsf{poly}(n)$ iterations we get an error probability that is negligible in n.

Testing primality. The problem of efficiently determining whether a given number p is prime has a long history. In the 1970s the first efficient *probabilistic* algorithms for testing primality were developed, and efficient algorithms

with the following property where shown: if the given input p is a prime number, then the output is always "prime". On the other hand, if p is a composite number, then the output is "composite" except with probability that is negligible in the length of p. Put differently, this means that if the result is "composite" then p is definitely composite, but if the output is "prime" then it is very likely that p is prime but it is also possible that a mistake has occurred (and p is actually composite).[2]

When using a randomized primality test of this sort in Algorithm 7.31 (the prime-generation algorithm shown earlier), the output of the algorithm is a random prime of the desired length as long as the algorithm does not output fail *and* the randomized primality test is always correct. This means that an additional source of error (besides the possibility of outputting fail) is introduced, and the algorithm may now output a composite number by mistake. Since we can ensure that this happens with only negligible probability, this remote possibility will be of no practical concern and we can safely ignore it.

A *deterministic* polynomial-time algorithm for testing primality was demonstrated in a breakthrough result in 2002. This algorithm, though running in polynomial time, is slower than the probabilistic tests mentioned above. For this reason, probabilistic primality tests are still used exclusively in practice for generating large primes.

In Section 7.2.2 we describe and analyze one of the most commonly-used probabilistic primality tests: the *Miller-Rabin* algorithm. This algorithm takes two inputs: an integer N being tested for primality and a parameter t that determines the error probability. The Miller-Rabin algorithm runs in time polynomial in $\|N\|$ and t, and satisfies:

THEOREM 7.33 *If N is prime, then the Miller-Rabin test always outputs "prime". If N is composite, then the algorithm outputs "prime" with probability at most 2^{-t} (and outputs the correct answer "composite" with probability $1 - 2^{-t}$).*

Putting it all together. Given the preceding discussion, we can now describe a polynomial-time prime-generation algorithm that, on input n, outputs a random n-bit prime except with probability negligible in n. (In the algorithm, c is the unspecified constant from Theorem 7.32.) The full procedure is described below in Algorithm 7.34.

Generating primes of a particular form. It is often desirable to generate a random n-bit prime p of a particular form, for example satisfying $p = 3 \bmod 4$ or such that $p = 2q + 1$ where q is also prime (p of the latter type are

[2] There also exist probabilistic primality tests that work in the opposite way: they always correctly identify composite numbers but sometimes make a mistake when given a prime as input. We will not consider algorithms of this type.

ALGORITHM 7.34
Generating a random prime

Input: A length parameter n
Output: A random n-bit prime

for $i = 1$ to n^2/c: {
  $p' \leftarrow \{0,1\}^{n-1}$
  $p := 1 \| p'$
  run the Miller-Rabin test on input p and parameter n
  if the output is "prime", **return** p
  }
return fail

called *strong primes*). In this case, appropriate modifications of the prime-generation algorithm shown above can be used (e.g., in order to obtain a prime of the form $p = 2q + 1$, generate a random prime q, compute $p = 2q + 1$ and output p if it too is prime). While these modified algorithms work well in practice, rigorous proofs that they run in polynomial time and fail with only negligible probability are more complex (and, in some cases, rely on unproven number-theoretic conjectures regarding the density of primes of a particular form). A detailed exploration of these issues is beyond the scope of this book, and we will simply assume the existence of appropriate prime-generation algorithms when needed.

7.2.2 * Primality Testing

We now describe the Miller-Rabin primality testing algorithm and prove Theorem 7.33. This material is not used directly in the rest of the book.

The key to the Miller-Rabin algorithm is to find a property that distinguishes primes and composites. As a starting point in this direction, consider the following observation: if N is prime then $|\mathbb{Z}_N^*| = N - 1$, and so for any number $a \in \{1, \ldots, N-1\}$ we have $a^{N-1} = 1 \bmod N$ by Theorem 7.14. This suggests testing whether a given integer N is prime by choosing a random element a and checking whether $a^{N-1} \stackrel{?}{=} 1 \bmod N$. If $a^{N-1} \neq 1 \bmod N$, then N cannot be prime. Conversely, we might hope that if N is not prime then there is a reasonable chance that we will pick a with $a^{N-1} \neq 1 \bmod N$, and so by repeating this test many times we could determine whether N is prime or not with high confidence. The above approach is shown as Algorithm 7.35. (Recall that exponentiation modulo N and computation of greatest common divisors can be carried out in polynomial time. Choosing a random element of $\{1, \ldots, N-1\}$ can also be done in polynomial time. See Appendix B.2.)

If N is prime then the discussion above implies that the algorithm always outputs "prime." If N is composite, the algorithm outputs "composite" if it finds an $a \in \mathbb{Z}_N^*$ such that $a^{N-1} \neq 1 \bmod N$ in any iteration. (It also outputs "composite" if it ever finds an $a \notin \mathbb{Z}_N^*$; we will take this into account later.)

ALGORITHM 7.35
Primality testing — first attempt

Input: Integer N and parameter t
Output: A decision as to whether N is prime or composite

for $i = 1$ to t:
  $a \leftarrow \{1, \ldots, N-1\}$
  if $\gcd(a, N) \neq 1$ **return** "composite"
  if $a^{N-1} \neq 1 \bmod N$ **return** "composite"
return "prime"

We refer to an $a \in \mathbb{Z}_N^*$ with this property as a *witness that N is composite*, or simply a *witness*. We might hope that when N is composite there are *many* witnesses, and thus the algorithm finds such a witness with "high" probability. This intuition is correct *provided there is at least one witness in the first place*. Before proving this, we need two group-theoretic lemmas.

PROPOSITION 7.36 *Let $\mathbb{G}$ be a finite group, and $\mathbb{H} \subseteq \mathbb{G}$. Assume that $\mathbb{H}$ contains the identity element of $\mathbb{G}$, and that for all $a, b \in \mathbb{H}$ it holds that $ab \in \mathbb{H}$. Then $\mathbb{H}$ is a subgroup of $\mathbb{G}$.*

PROOF We need to verify that $\mathbb{H}$ satisfies all the conditions of Definition 7.9. Associativity in $\mathbb{H}$ is inherited automatically from $\mathbb{G}$. By assumption, $\mathbb{H}$ has the identity element and is closed under the group operation. The only thing remaining to verify is that the inverse of every element in $\mathbb{H}$ also lies in $\mathbb{H}$. Let m be the order of $\mathbb{G}$ (here is where we use the fact that $\mathbb{G}$ is finite), and consider an arbitrary element $a \in \mathbb{H}$. Since $a \in \mathbb{G}$, we have $1 = a^m = a \cdot a^{m-1}$. This means that a^{m-1} is the inverse of a. Since $a \in \mathbb{H}$, the closure property of $\mathbb{H}$ guarantees that $a^{m-1} \in \mathbb{H}$ as required. ■

LEMMA 7.37 *Let $\mathbb{H}$ be a strict subgroup of a finite group $\mathbb{G}$ (i.e., $\mathbb{H} \neq \mathbb{G}$). Then $|\mathbb{H}| \leq |\mathbb{G}|/2$.*

PROOF Let $\bar{h}$ be an element of $\mathbb{G}$ that is *not* in $\mathbb{H}$; since $\mathbb{H} \neq \mathbb{G}$, we know such an $\bar{h}$ exists. Consider the set $\bar{\mathbb{H}} \stackrel{\text{def}}{=} \{\bar{h}h \mid h \in \mathbb{H}\}$. We show that (1) $|\bar{\mathbb{H}}| = |\mathbb{H}|$, and (2) every element of $\bar{\mathbb{H}}$ lies outside of $\mathbb{H}$; i.e., the intersection of $\mathbb{H}$ and $\bar{\mathbb{H}}$ is empty. Since both $\mathbb{H}$ and $\bar{\mathbb{H}}$ are subsets of $\mathbb{G}$, these imply $|\mathbb{G}| \geq |\mathbb{H}| + |\bar{\mathbb{H}}| = 2|\mathbb{H}|$, proving the lemma.

For every $h_1, h_2 \in \mathbb{H}$, if $\bar{h}h_1 = \bar{h}h_2$ then, multiplying by $\bar{h}^{-1}$ on each side, we have $h_1 = h_2$. This shows that every distinct element $h \in \mathbb{H}$ corresponds to a distinct element $\bar{h}h \in \bar{\mathbb{H}}$, proving (1).

Assume toward a contradiction that $\bar{h}h \in \mathbb{H}$ for some h. This means $\bar{h}h = h'$ for some $h' \in \mathbb{H}$, and so $\bar{h} = h'h^{-1}$. Now, $h'h^{-1} \in \mathbb{H}$ since $\mathbb{H}$ is a subgroup

and $h', h^{-1} \in \mathbb{H}$. But this means that $\bar{h} \in \mathbb{H}$, in contradiction to the way $\bar{h}$ was chosen. This proves (2), and completes the proof of the lemma. ■

The following theorem will enable us to analyze the algorithm given earlier.

THEOREM 7.38 *Fix N. Say there exists a witness that N is composite. Then at least half the elements of $\mathbb{Z}_N^*$ are witnesses that N is composite.*

PROOF Let Bad be the set of elements in $\mathbb{Z}_N^*$ that are *not* witnesses; that is, $a \in \mathsf{Bad}$ means $a^{N-1} = 1 \bmod N$. Clearly, $1 \in \mathsf{Bad}$. If $a, b \in \mathsf{Bad}$, then $(ab)^{N-1} = a^{N-1} \cdot b^{N-1} = 1 \cdot 1 = 1 \bmod N$ and hence $ab \in \mathsf{Bad}$. By Lemma 7.36, we conclude that Bad is a subgroup of $\mathbb{Z}_N^*$. Since (by assumption) there is at least one witness, Bad is a *strict* subgroup of $\mathbb{Z}_N^*$. Lemma 7.37 then shows that $|\mathsf{Bad}| \leq |\mathbb{Z}_N^*|/2$, showing that at least half the elements of $\mathbb{Z}_N^*$ are *not* in Bad (and hence are witnesses). ■

Let N be composite. If there exists a witness that N is composite, then there are at least $|\mathbb{Z}_N^*|/2$ witnesses. The probability that we find either a witness or an element not in $\mathbb{Z}_N^*$ in any given iteration of the algorithm is thus at least

$$\frac{\frac{|\mathbb{Z}_N^*|}{2} + \Big((N-1) - |\mathbb{Z}_N^*|\Big)}{N-1} = 1 - \frac{|\mathbb{Z}_N^*|/2}{(N-1)} \geq 1 - \frac{|\mathbb{Z}_N^*|/2}{|\mathbb{Z}_N^*|} = \frac{1}{2},$$

and so the probability that the algorithm does not find a witness in any of the t iterations (and hence the probability that the algorithm mistakenly outputs "prime") is at most 2^{-t}.

The above, unfortunately, does not give a complete solution since there are infinitely-many composite numbers N that do not have *any* witnesses that they are composite! Such values N are known as *Carmichael numbers*; a detailed discussion is beyond the scope of this book.

Happily, a refinement of the above test can be shown to work for all N. Let $N - 1 = 2^r u$, where u is odd and $r \geq 1$. (It is easy to compute r and u given N. Also, restricting to $r \geq 1$ means that N is odd, but testing primality is easy when N is even!) The algorithm shown previously tests only whether $a^{N-1} = a^{2^r u} = 1 \bmod N$. A more refined algorithm looks at the *sequence* of $r+1$ values $a^u, a^{2u}, \ldots, a^{2^r u}$ (all modulo N). Each term in this sequence is the square of the preceding term; thus, if some value is equal to ± 1 then all subsequent values will be equal to 1.

Say that $a \in \mathbb{Z}_N^*$ is a *strong witness that N is composite* (or simply a *strong witness*) if (1) $a^u \neq \pm 1 \bmod N$ and (2) $a^{2^i u} \neq -1 \bmod N$ for all

$i \in \{1, \ldots, r-1\}$. If a is *not* a strong witness then $a^{2^{r-1}u} = \pm 1 \bmod N$ and

$$a^{N-1} = a^{2^r u} = \left(a^{2^{r-1}u}\right)^2 = 1 \bmod N,$$

and so a is not a witness that N is composite, either. Put differently, if a is a witness then it is also a strong witness and so there can only possibly be *more* strong witnesses than witnesses. Note also that when an element a is *not* a strong witness then the sequence $(a^u, a^{2u}, \ldots, a^{2^r u})$ (all taken modulo N) takes one of the following forms:

$$(\pm 1, 1, \ldots, 1) \quad \text{or} \quad (\star, \ldots, \star, -1, 1, \ldots, 1),$$

where $\star$ denotes an arbitrary term.

We first show that if N is prime then there does not exist a strong witness that N is composite. In doing so, we rely on the following easy lemma (which is a special case of Proposition 11.1 proved in Chapter 11):

LEMMA 7.39 *Say $x \in \mathbb{Z}_N^*$ is a* square root of 1 modulo N *if $x^2 = 1 \bmod N$. If N is an odd prime then the only square roots of 1 modulo N are $[\pm 1 \bmod N]$.*

PROOF Clearly $(\pm 1)^2 = 1 \bmod N$. Now, say N is an odd prime and $x^2 = 1 \bmod N$ with $x \in \{1, \ldots, N-1\}$. Then $0 = x^2 - 1 = (x+1)(x-1) \bmod N$, implying that $N \,|\, (x+1)$ or $N \,|\, (x-1)$ by Proposition 7.3. This can only possibly occur if $x = [\pm 1 \bmod N]$. ∎

Now, say N is an odd prime and fix arbitrary $a \in \mathbb{Z}_N^*$. Let $i \geq 0$ be the minimum value for which $a^{2^i u} = 1 \bmod N$; since $a^{2^r u} = a^{N-1} = 1 \bmod N$ we know that some such $i \leq r$ exists. If $i = 0$ then $a^u = 1 \bmod N$ and a is not a strong witness. Otherwise,

$$\left(a^{2^{i-1}u}\right)^2 = a^{2^i u} = 1 \bmod N$$

and $a^{2^{i-1}u}$ is a square root of 1. If N is an odd prime, the only square roots of 1 are ± 1; by choice of i, however, $a^{2^{i-1}u} \neq 1 \bmod N$. So $a^{2^{i-1}u} = -1 \bmod N$, and a is not a strong witness. We conclude that when N is an odd prime there is no strong witness for N.

A composite integer N is a *prime power* if $N = \hat{p}^e$ for some prime $\hat{p}$ and integer $e \geq 2$. We now show that every odd composite N that is not a prime power has "many" strong witnesses.

THEOREM 7.40 *Let N be an odd, composite number that is not a prime power. Then at least half the elements of $\mathbb{Z}_N^*$ are strong witnesses that N is composite.*

PROOF Let $\mathsf{Bad} \subseteq \mathbb{Z}_N^*$ denote the set of elements that are not strong witnesses. We define a set Bad' and show that: (1) Bad is a subset of Bad', and (2) Bad' is a strict subgroup of $\mathbb{Z}_N^*$. This suffices because by combining (2) and Lemma 7.37 we have that $|\mathsf{Bad}'| \leq |\mathbb{Z}_N^*|/2$. Furthermore, by (1) it holds that $\mathsf{Bad} \subseteq \mathsf{Bad}'$, and so $|\mathsf{Bad}| \leq |\mathsf{Bad}'| \leq |\mathbb{Z}_N^*|/2$ as in Theorem 7.38. Thus, at least half the elements of $\mathbb{Z}_N^*$ are strong witnesses. (We stress that we do not claim that Bad is a subgroup of $\mathbb{Z}_N^*$.)

Note first that $-1 \in \mathsf{Bad}$ since $(-1)^u = -1 \bmod N$ (recall u is odd). Let $i \in \{0, \ldots, r-1\}$ be the largest integer for which there exists an $a \in \mathsf{Bad}$ with $a^{2^i u} = -1 \bmod N$; alternatively, i is the largest integer for which there exists an $a \in \mathsf{Bad}$ with

$$(a^u, a^{2u}, \ldots, a^{2^r u}) = (\underbrace{\star, \ldots, \star, -1}_{i+1 \text{ terms}}, 1, \ldots, 1).$$

Since $-1 \in \mathsf{Bad}$ and $(-1)^{2^0 u} = -1 \bmod N$, such i is well-defined.

Fix i as above, and define

$$\mathsf{Bad}' \stackrel{\text{def}}{=} \{a \mid a^{2^i u} = \pm 1 \bmod N\}.$$

We now prove what we claimed above.

CLAIM 7.41 $\mathsf{Bad} \subseteq \mathsf{Bad}'$.

Let $a \in \mathsf{Bad}$. Then either $a^u = 1 \bmod N$ or $a^{2^j u} = -1 \bmod N$ for some $j \in \{0, \ldots, r-1\}$. In the first case, $a^{2^i u} = (a^u)^{2^i} = 1 \bmod N$ and so $a \in \mathsf{Bad}'$. In the second case, we have $j \leq i$ by choice of i. If $j = i$ then clearly $a \in \mathsf{Bad}'$. If $j < i$ then $a^{2^i u} = (a^{2^j u})^{2^{i-j}} = 1 \bmod N$ and $a \in \mathsf{Bad}'$. Since a was arbitrary, this shows $\mathsf{Bad} \subseteq \mathsf{Bad}'$.

CLAIM 7.42 Bad' *is a subgroup of* $\mathbb{Z}_N^*$.

Clearly $1 \in \mathsf{Bad}'$. Furthermore, if $a, b \in \mathsf{Bad}'$ then

$$(ab)^{2^i u} = a^{2^i u} b^{2^i u} = (\pm 1)(\pm 1) = \pm 1 \bmod N$$

and so $ab \in \mathsf{Bad}'$. By Lemma 7.36, Bad' is a subgroup.

CLAIM 7.43 Bad' *is a strict subgroup of* $\mathbb{Z}_N^*$.

If N is a composite integer that is not a prime power, then N can be written as $N = N_1 N_2$ with $\gcd(N_1, N_2) = 1$. Appealing to the Chinese remainder theorem, let the notation $a \leftrightarrow (a_1, a_2)$ denote the representation of $a \in \mathbb{Z}_N^*$ as an element of $\mathbb{Z}_{N_1}^* \times \mathbb{Z}_{N_2}^*$; that is, $a_1 = [a \bmod N_1]$ and $a_2 = [a \bmod N_2]$.

Take $a \in \mathsf{Bad}'$ such that $a^{2^i u} = -1 \bmod N$ (such an a must exist by the way we defined i), and say $a \leftrightarrow (a_1, a_2)$. We know that

$$(a_1^{2^i u}, a_2^{2^i u}) = (a_1, a_2)^{2^i u} \leftrightarrow a^{2^i u} = -1 \leftrightarrow (-1, -1),$$

and so

$$a_1^{2^i u} = -1 \bmod N_1 \quad \text{and} \quad a_2^{2^i u} = -1 \bmod N_2.$$

Consider the element $b \in \mathbb{Z}_N^*$ with $b \leftrightarrow (a_1, 1)$. Then

$$b^{2^i u} \leftrightarrow (a_1, 1)^{2^i u} = (a_1^{2^i u}, 1) = (-1, 1) \not\leftrightarrow \pm 1.$$

That is, $b^{2^i u} \neq \pm 1 \bmod N$ and so we have found an element $b \notin \mathsf{Bad}'$. As we have mentioned, this proves that Bad' is a *strict* subgroup of $\mathbb{Z}_N^*$ and so, by Lemma 7.37, the size of Bad' (and thus the size of Bad) is at most half the size of $\mathbb{Z}_N^*$, as required. ■

An integer N is a *perfect power* if $N = \hat{N}^e$ for integers $\hat{N}$ and $e \geq 2$ (here it is not required for $\hat{N}$ to be prime, though of course any prime power is also a perfect power). We can now describe a primality testing algorithm in full.

ALGORITHM 7.44
The Miller-Rabin primality test

Input: Integer $N > 2$ and parameter t
Output: A decision as to whether N is prime or composite

if N is even, **return** "composite"
if N is a perfect power, **return** "composite"
compute $r \geq 1$ and u odd such that $N - 1 = 2^r u$
for $j = 1$ to t:
  $a \leftarrow \{1, \ldots, N-1\}$
  if $\gcd(a, N) \neq 1$ **return** "composite"
  if a is a strong witness **return** "composite"
return "prime"

Exercises 7.11 and 7.12 ask you to show that testing whether N is a perfect power, and testing whether a particular a is a strong witness, can be done in polynomial time. Given these results, the algorithm clearly runs in time polynomial in $\|N\|$ and t. We can now complete the proof of Theorem 7.33.

PROOF If N is prime, there are no strong witnesses and so the Miller-Rabin algorithm always outputs "prime". If N is composite there are two cases: if N is a prime power the algorithm always outputs "composite". Otherwise, we invoke Theorem 7.40 and see that, in any iteration, the probability

of finding either a strong witness or an element not in $\mathbb{Z}_N^*$ is at least

$$\frac{|\mathbb{Z}_N^*|/2 + \left((N-1) - |\mathbb{Z}_N^*|\right)}{N-1} = 1 - \frac{|\mathbb{Z}_N^*|/2}{N-1} \geq 1 - \frac{|\mathbb{Z}_N^*|/2}{|\mathbb{Z}_N^*|} = \frac{1}{2},$$

and so the probability that the algorithm does not find a witness in any of the t iterations (and hence outputs "prime") is at most 2^{-t}. ■

7.2.3 The Factoring Assumption

Now that we have discussed how to generate random primes, we formally define the factoring assumption. Let GenModulus be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$, and p and q are n-bit primes except with probability negligible in n. Then consider the following experiment for a given algorithm $\mathcal{A}$ and parameter n:

The factoring experiment $\mathsf{Factor}_{\mathcal{A},\mathsf{GenModulus}}(n)$**:**

1. *Run* $\mathsf{GenModulus}(1^n)$ *to obtain* (N, p, q).
2. $\mathcal{A}$ *is given* N*, and outputs* $p', q' > 1$.
3. *The output of the experiment is defined to be 1 if* $p' \cdot q' = N$*, and 0 otherwise.*

Of course, except with negligible probability, if the output of the experiment is 1 then $\{p', q'\} = \{p, q\}$.

DEFINITION 7.45 *We say that* factoring is hard relative to GenModulus *if for all probabilistic polynomial-time algorithms* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Factor}_{\mathcal{A},\mathsf{GenModulus}}(n) = 1] \leq \mathsf{negl}(n).$$

The factoring assumption is simply the assumption that there exists a GenModulus relative to which factoring is hard. A natural way to construct a suitable GenModulus algorithm is to generate two random primes p and q of length n, and then set N to be their product; factoring is believed to be hard relative to GenModulus of this form.

7.2.4 The RSA Assumption

The factoring problem has been studied for hundreds of years without an efficient algorithm being found, and so it is very plausible that the problem truly is hard. Unfortunately, although the factoring assumption does yield a

one-way function (see Section 7.4.1), the factoring assumption *in the form we have described it* is not known to yield practical cryptographic constructions. (In Section 11.2.2, however, we show a very useful problem whose hardness is *equivalent* to that of factoring.) This has motivated a search for other problems whose difficulty is related to the hardness of factoring. The best known of these is a problem introduced by Rivest, Shamir, and Adleman and now called the *RSA problem*.

$\mathbb{Z}_N^*$ is a group of order $\phi(N) = (p-1)(q-1)$. If the factorization of N is known, then it is easy to compute the group order $\phi(N)$ and so computations modulo N can potentially be simplified by "working in the exponent modulo $\phi(N)$" (cf. Corollary 7.15). On the other hand, if the factorization of N is unknown then it is difficult to compute $\phi(N)$ (in fact, computing $\phi(N)$ is as hard as factoring N; see Exercise 7.13). Thus "working in the exponent modulo $\phi(N)$" is not an available option, at least not in any obvious way. The RSA problem exploits this asymmetry: the RSA problem is easy to solve if $\phi(N)$ is known, but appears hard to solve without knowledge of $\phi(N)$. In this section we focus on the hardness of solving the RSA problem relative to a modulus N of unknown factorization; the fact that the RSA problem becomes easy when the factors of N are known will prove extremely useful for the cryptographic applications we will see later in the book.

Given a modulus N and an integer $e > 0$ relatively prime to $\phi(N)$, Corollary 7.22 shows that exponentiation to the eth power modulo N is a *permutation*. It therefore makes sense to define $y^{1/e} \bmod N$ (for any $y \in \mathbb{Z}_N^*$) as the unique element of $\mathbb{Z}_N^*$ for which $\left(y^{1/e}\right)^e = y \bmod N$.

The RSA problem can now be described informally as follows: given N, an integer $e > 0$ that is relatively prime to $\phi(N)$, and an element $y \in \mathbb{Z}_N^*$, compute $y^{1/e} \bmod N$; that is, given N, e, y find x such that $x^e = y \bmod N$. Formally, let GenRSA be a probabilistic polynomial-time algorithm that, on input 1^n, outputs a modulus N that is the product of two n-bit primes, as well as an integer $e > 0$ with $\gcd(e, \phi(N)) = 1$ and an integer d satisfying $ed = 1 \bmod \phi(N)$. (Such a d exists since e is invertible modulo $\phi(N)$.) The algorithm may fail with probability negligible in n. Consider the following experiment for a given algorithm $\mathcal{A}$ and parameter n:

The RSA experiment $\mathsf{RSA\text{-}inv}_{\mathcal{A},\mathsf{GenRSA}}(n)$**:**

1. *Run* $\mathsf{GenRSA}(1^n)$ *to obtain* (N, e, d).
2. *Choose* $y \leftarrow \mathbb{Z}_N^*$.
3. $\mathcal{A}$ *is given* N, e, y, *and outputs* $x \in \mathbb{Z}_N^*$.
4. *The output of the experiment is defined to be 1 if* $x^e = y \bmod N$, *and 0 otherwise.*

DEFINITION 7.46 *We say that* the RSA problem is hard relative to GenRSA *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A},\mathsf{GenRSA}}(n) = 1] \leq \mathsf{negl}(n).$$

The RSA assumption is simply the assumption that there exists a GenRSA relative to which the RSA problem is hard. A suitable algorithm GenRSA can be constructed based on any algorithm GenModulus that generates a composite modulus along with its factorization. A high-level outline follows, where the only thing left unspecified is how exactly e is chosen. There are in fact a number of ways e can be chosen (with the RSA problem still believed to be hard); some specific methods for choosing e are discussed in Section 10.4.1.

ALGORITHM 7.47
GenRSA — **high-level outline**

Input: Security parameter 1^n
Output: N, e, d as described in the text

$(N, p, q) \leftarrow \mathsf{GenModulus}(1^n)$
$\phi(N) := (p-1)(q-1)$
find e such that $\gcd(e, \phi(N)) = 1$
compute $d := [e^{-1} \bmod \phi(N)]$
return N, e, d

When GenRSA is constructed as above, for which algorithms GenModulus is the RSA problem likely to be hard? If the factorization of N is known, the RSA problem is easy to solve: first compute $\phi(N)$; then compute $d = [e^{-1} \bmod \phi(N)]$; finally compute the solution $[y^d \bmod N]$. It follows from Corollary 7.22 that this gives the correct answer. For the RSA problem to be hard, then, it must be infeasible to factor N output by GenModulus. We conclude that if the RSA problem is hard relative to GenRSA constructed as above, then the factoring problem must be hard relative to GenModulus. That is, the RSA problem cannot be *more* difficult than factoring.

What about the converse? When N is a product of two primes, the factorization of N can be computed efficiently from $\phi(N)$ (see Exercise 7.13) and so the problems of factoring N and computing $\phi(N)$ are *equally hard*. In fact, one can show more: given N, e, and d with $ed = 1 \bmod \phi(N)$ it is possible to compute the factorization of N in probabilistic polynomial time; see Exercise 7.14 for a simple case of this result. There is no known proof, however, that there is no *other* way of solving the RSA problem that does not involve explicit computation of $\phi(N)$ or d. Thus, given our current state of knowledge, we cannot conclude that the RSA problem is as hard as factoring, and so the assumption that RSA is hard appears stronger than the assump-

tion that factoring is hard. (That is, it may be that the RSA problem can be solved in polynomial time even though factoring cannot.)

Nevertheless, when GenRSA is constructed based on a modulus-generation algorithm GenModulus as in Algorithm 7.47 (i.e., by choosing N as the product of two random n-bit primes), the RSA problem is believed to be hard relative to GenRSA whenever factoring is hard relative to GenModulus.

7.3 Assumptions in Cyclic Groups

In this section we introduce a class of cryptographic hardness assumptions in *cyclic groups*. We first discuss the necessary background.

7.3.1 Cyclic Groups and Generators

Let $\mathbb{G}$ be a finite group of order m. For arbitrary $g \in \mathbb{G}$, consider the set

$$\langle g \rangle \stackrel{\text{def}}{=} \{g^0, g^1, \ldots, \}.$$

By Theorem 7.14, we have $g^m = 1$. Let $i \leq m$ be the smallest positive integer for which $g^i = 1$. Then the above sequence repeats after i terms (i.e., $g^i = g^0$, $g^{i+1} = g^1$, etc.), and so

$$\langle g \rangle = \{g^0, \ldots, g^{i-1}\}.$$

We see that $\langle g \rangle$ contains at most i elements. In fact, it contains exactly i elements since if $g^j = g^k$ with $0 \leq j < k < i$ then $g^{k-j} = 1$ and $0 < k - j < i$, contradicting our choice of i.

It is not hard to verify that $\langle g \rangle$ is a subgroup of $\mathbb{G}$ for any g (see Exercise 7.3); we call $\langle g \rangle$ the *subgroup generated by* g. If the order of the subgroup $\langle g \rangle$ is i, then i is called the *order of* g; that is:

DEFINITION 7.48 *Let $\mathbb{G}$ be a finite group and $g \in \mathbb{G}$. The* order of g *is the smallest positive integer i with $g^i = 1$.*

The following is a useful analogue of Corollary 7.15 (the proof is identical):

PROPOSITION 7.49 *Let $\mathbb{G}$ be a finite group, and $g \in \mathbb{G}$ an element of order i. Then for any integer x, we have $g^x = g^{[x \bmod i]}$.*

We can actually prove something stronger.

PROPOSITION 7.50 *Let $\mathbb{G}$ be a finite group, and $g \in \mathbb{G}$ an element of order i. Then $g^x = g^y$ if and only if $x = y \bmod i$.*

PROOF If $x = y \bmod i$ then $[x \bmod i] = [y \bmod i]$ and the previous proposition says that

$$g^x = g^{[x \bmod i]} = g^{[y \bmod i]} = g^y.$$

For the more interesting direction, say $g^x = g^y$. Let $x' = [x \bmod i]$ and $y' = [y \bmod i]$; the previous proposition tells us that $g^{x'} = g^{y'}$ or, equivalently, $g^{x'}(g^{y'})^{-1} = 1$. If $x' \neq y'$, we may assume without loss of generality that $x' > y'$. Since both x' and y' are smaller than i, the difference $x' - y'$ is then a non-zero integer smaller than i. But then

$$1 = g^{x'} \cdot \left(g^{y'}\right)^{-1} = g^{x'-y'},$$

contradicting the fact that i is the order of g. ■

The identity element of any group $\mathbb{G}$ has order 1, generates the group $\langle 1 \rangle = \{1\}$, and is the only element of order 1. At the other extreme, if there exists an element $g \in \mathbb{G}$ that has order m (where m is the order of $\mathbb{G}$), then $\langle g \rangle = \mathbb{G}$. In this case, we call $\mathbb{G}$ a *cyclic group* and say that g is a *generator* of $\mathbb{G}$. (Note that a cyclic group may have multiple generators, and so we cannot speak of *the* generator.) If g is a generator of $\mathbb{G}$ then, by definition, every element $h \in \mathbb{G}$ is equal to g^x for some $x \in \{0, \ldots, m-1\}$, a point we will return to in the next section.

Different elements of the same group $\mathbb{G}$ may have different orders. We can, however, place some restrictions on what these possible orders might be.

PROPOSITION 7.51 *Let $\mathbb{G}$ be a finite group of order m, and say $g \in \mathbb{G}$ has order i. Then $i \mid m$.*

PROOF By Theorem 7.14 we know that $g^m = 1$. Since g has order i, we have $g^m = g^{[m \bmod i]}$ by Proposition 7.49. If i does not divide m, then $i' \stackrel{\text{def}}{=} [m \bmod i]$ is a positive integer smaller than i for which $g^{i'} = 1$. Since i is the order of g, this is impossible. ■

The next corollary illustrates the power of this result:

COROLLARY 7.52 *If $\mathbb{G}$ is a group of prime order p, then $\mathbb{G}$ is cyclic. Furthermore, all elements of $\mathbb{G}$ except the identity are generators of $\mathbb{G}$.*

PROOF By Proposition 7.51, the only possible orders of elements in $\mathbb{G}$ are 1 and p. Only the identity has order 1, and so all other elements have order p and generate $\mathbb{G}$. ■

Groups of prime order form one class of cyclic groups. The additive group $\mathbb{Z}_N$, for $N > 1$, gives another example of a cyclic group (the element 1 is always a generator). The next theorem gives an important additional class of cyclic groups; a proof is outside the scope of this book, but can be found in any standard abstract algebra text.

THEOREM 7.53 *If p is prime then $\mathbb{Z}_p^*$ is cyclic.*

For $p > 3$ prime, $\mathbb{Z}_p^*$ does not have prime order and so the above does not follow from the preceding corollary.

Some examples will help illustrate the preceding discussion.

Example 7.54

Consider the (additive) group $\mathbb{Z}_{15}$. As we have noted, $\mathbb{Z}_{15}$ is cyclic and the element '1' is a generator since $15 \cdot 1 = 0 \bmod 15$ and $i' \cdot 1 = i' \neq 0 \bmod 15$ for any $0 < i' < 15$ (recall that in this group the identity is 0).

$\mathbb{Z}_{15}$ has other generators. E.g., $\langle 2 \rangle = \{0, 2, 4, \ldots, 14, 1, 3, 5, \ldots, 13\}$ and so 2 is also a generator.

Not every element generates $\mathbb{Z}_{15}$. For example, the element '3' has order 5 since $5 \cdot 3 = 0 \bmod 15$, and so 3 does not generate $\mathbb{Z}_{15}$. The subgroup $\langle 3 \rangle$ consists of the 5 elements $\{0, 3, 6, 9, 12\}$, and this is indeed a subgroup under addition modulo 15. The element '10' has order 3 since $3 \cdot 10 = 0 \bmod 15$, and the subgroup $\langle 10 \rangle$ consists of the 3 elements $\{0, 5, 10\}$. Note that 5 and 3 both divide $|\mathbb{Z}_{15}| = 15$ as required by Proposition 7.51. ♢

Example 7.55

Consider the (multiplicative) group $\mathbb{Z}_{15}^*$ of order $(5-1)(3-1) = 8$. We have $\langle 2 \rangle = \{1, 2, 4, 8\}$, and so the order of 2 is 4. As required by Proposition 7.51, 4 divides 8. ♢

Example 7.56

Consider the group $\mathbb{Z}_p$ of prime order p. We know this group is cyclic, but Corollary 7.52 tells us more: namely, that *every* element except 0 is a generator. Indeed, for any element $h \in \{1, \ldots, p-1\}$ and integer $i > 0$ we have $ih = 0 \bmod p$ if and only if $p \,|\, ih$. But then Proposition 7.3 says that either $p \,|\, h$ or $p \,|\, i$. The former cannot occur (since $h < p$), and the smallest positive integer for which the latter can occur is $i = p$. We have thus shown that every non-zero element h has order p (and so generates $\mathbb{Z}_p$), in accordance with Corollary 7.52. ♢

Example 7.57
Consider the group $\mathbb{Z}_7^*$, which is cyclic by Theorem 7.53. We have $\langle 2 \rangle = \{1, 2, 4\}$, and so 2 is *not* a generator. However,

$$\langle 3 \rangle = \{1, 3, 2, 6, 4, 5\} = \mathbb{Z}_7^*,$$

and so 3 is a generator of $\mathbb{Z}_7^*$. ♢

The following example relies on the material of Section 7.1.5.

Example 7.58
Let $\mathbb{G}$ be a cyclic group of order n, and let g be a generator of $\mathbb{G}$. Then the mapping $f : \mathbb{Z}_n \to \mathbb{G}$ given by $f(a) = g^a$ is an isomorphism between $\mathbb{Z}_n$ and $\mathbb{G}$. Indeed, for $a, a' \in \mathbb{Z}_n$ we have

$$f(a + a') = g^{[a+a' \bmod n]} = g^{a+a'} = g^a \cdot g^{a'} = f(a) \cdot f(a').$$

Bijectivity of f can be proved using the fact that n is the order of g. ♢

The previous example shows that all cyclic groups of the same order are "the same" in an *algebraic* sense. We stress that this is not true in a *computational* sense, and in particular an isomorphism $f^{-1} : \mathbb{G} \to \mathbb{Z}_n$ (which we know must exist) need not be efficiently computable. Moreover, even though $\mathbb{Z}_p^*$ (for p prime) is isomorphic to the group $\mathbb{Z}_{p-1}$, the computational complexity of operations in these two groups may be very different. We will return to this point in Chapter 8.

7.3.2 The Discrete Logarithm and Diffie-Hellman Assumptions

We now introduce a number of computational problems that can be defined for any class of cyclic groups. We will keep the discussion in this section abstract, and consider specific examples of groups in which these problems are believed to be hard in Sections 7.3.3 and 7.3.4.

If $\mathbb{G}$ is a cyclic group of order q, then there exists a generator $g \in \mathbb{G}$ such that $\{g^0, g^1, \ldots, g^{q-1}\} = \mathbb{G}$. Equivalently, for every $h \in \mathbb{G}$ there is a *unique* $x \in \mathbb{Z}_q$ such that $g^x = h$. By way of notation, when the underlying group $\mathbb{G}$ is understood from the context we call this x the *discrete logarithm of h with respect to g* and write $x = \log_g h$. Note that if $g^{x'} = h$ for some arbitrary integer x', then $\log_g h = [x' \bmod q]$. We remark that logarithms in this case are called "discrete" since they take values in a finite range, as opposed to "standard" logarithms from calculus whose values range over an infinite set.

Discrete logarithms obey many of the same rules as "standard" logarithms. For example, $\log_g 1 = 0$ (where '1' is the identity of $\mathbb{G}$) and $\log_g(h_1 \cdot h_2) = [(\log_g h_1 + \log_g h_2) \bmod q]$.

The *discrete logarithm problem* in a cyclic group $\mathbb{G}$ with given generator g is to compute $\log_g h$ given a random element $h \in \mathbb{G}$ as input. Formally, let $\mathcal{G}$ be a polynomial-time algorithm that, on input 1^n, outputs a (description of a) cyclic group $\mathbb{G}$, its order q (with $\|q\| = n$), and a generator $g \in \mathbb{G}$. We also require that the group operation in $\mathbb{G}$ can be computed efficiently (namely, in time polynomial in n). Consider the following experiment for a given group-generating algorithm $\mathcal{G}$, algorithm $\mathcal{A}$, and parameter n:

The discrete logarithm experiment $\mathsf{DLog}_{\mathcal{A},\mathcal{G}}(n)$:

1. *Run $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$, where $\mathbb{G}$ is a cyclic group of order q (with $\|q\| = n$), and g is a generator of $\mathbb{G}$.*
2. *Choose $h \leftarrow \mathbb{G}$. (This can be done by choosing $x' \leftarrow \mathbb{Z}_q$ and setting $h := g^{x'}$.)*
3. *$\mathcal{A}$ is given $\mathbb{G}, q, g, h$, and outputs $x \in \mathbb{Z}_q$.*
4. *The output of the experiment is defined to be 1 if $g^x = h$, and 0 otherwise.*

DEFINITION 7.59 *We say that* the discrete logarithm problem is hard relative to $\mathcal{G}$ *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\Pr[\mathsf{DLog}_{\mathcal{A},\mathcal{G}}(n) = 1] \leq \mathsf{negl}(n).$$

The discrete logarithm assumption is simply the assumption that there exists a $\mathcal{G}$ for which the discrete logarithm problem is hard. The following two sections discuss some candidate group-generation algorithms $\mathcal{G}$ for which this is believed to be the case.

Some very useful problems that are related to the problem of computing discrete logarithms are the so-called *Diffie-Hellman* problems. There are two important variants: the *computational* Diffie-Hellman (CDH) problem, and the *decisional* Diffie-Hellman (DDH) problem. Although the CDH problem is not used in the remainder of the book, it will be instructive to introduce it, at least informally, before moving on to the DDH problem.

Fix a cyclic group $\mathbb{G}$ and a generator $g \in \mathbb{G}$. Given two group elements h_1 and h_2, define $\mathsf{DH}_g(h_1, h_2) \stackrel{\text{def}}{=} g^{\log_g h_1 \cdot \log_g h_2}$. That is, if $h_1 = g^x$ and $h_2 = g^y$ then

$$\mathsf{DH}_g(h_1, h_2) = g^{x \cdot y} = h_1^y = h_2^x\,.$$

The *CDH problem* is to compute $\mathsf{DH}_g(h_1, h_2)$ given randomly-chosen h_1 and h_2.

If the discrete logarithm problem relative to some $\mathcal{G}$ is easy, then the CDH problem is, too: given h_1 and h_2, first compute $x = \log_g h_1$ and then output the answer h_2^x. In contrast, it is not clear whether hardness of the discrete logarithm problem necessarily implies that the CDH problem is hard as well.

The *DDH problem*, roughly speaking, is to distinguish $\mathsf{DH}_g(h_1, h_2)$ from a random group element for randomly-chosen h_1, h_2. That is, given randomly-chosen h_1, h_2 and a candidate solution h', the problem is to decide whether $h' = \mathsf{DH}_g(h_1, h_2)$ or whether h' was chosen randomly from $\mathbb{G}$. Formally, let $\mathcal{G}$ be as above. Then:

DEFINITION 7.60 *We say that* the DDH problem is hard relative to $\mathcal{G}$ *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\left| \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1] \right| \leq \mathsf{negl}(n),$$

where in each case the probabilities are taken over the experiment in which $\mathcal{G}(1^n)$ outputs $(\mathbb{G}, q, g)$, and then random $x, y, z \in \mathbb{Z}_q$ are chosen.

Note that when z is chosen at random from $\mathbb{Z}_q$, independent of anything else, the element g^z is uniformly distributed in $\mathbb{G}$.

We have already seen that if the discrete logarithm problem is easy relative to some $\mathcal{G}$, then the CDH problem is too. Similarly, if the CDH problem is easy relative to $\mathcal{G}$ then so is the DDH problem; you are asked to show this in Exercise 7.16. The converse, however, does not appear to be true, and there are examples of groups in which the discrete logarithm and CDH problems are believed to be hard even though the DDH problem is easy; see Exercise 11.10.

Using Prime-Order Groups

There are a number of classes of cyclic groups for which the discrete logarithm and Diffie-Hellman problems are believed to be hard. Although cyclic groups of non-prime order are still used for certain cryptographic applications, there is a general preference for using cyclic groups of *prime order*. There are a number of reasons for this, as we now explain.

One reason for preferring groups of prime order is because, in a certain sense, the discrete logarithm problem is hardest in such groups. Specifically, the *Pohlig-Hellman algorithm* that will be described in Chapter 8 reduces an instance of the discrete logarithm problem in a group of order $q = q_1 \cdot q_2$ to two instances of the discrete logarithm problem in groups of order q_1 and q_2, respectively. (This assumes that the factorization of q is known, but if q has small prime factors then finding some non-trivial factorization of q will be easy.) We stress that this does not mean that the discrete logarithm problem is *easy* (i.e., can be solved in polynomial time) in non-prime order groups; it merely means that the problem becomes *easier* (at least for currently known algorithms). In any case, this explains why prime order groups are desirable.

A second motivation for using prime order groups is because finding a generator in such groups is trivial, as is testing whether a given element is a

generator. This follows from Corollary 7.52, which says that *every* element of a prime order group (except the identity) is a generator. Even though it is possible to find a generator of an arbitrary cyclic group in probabilistic polynomial time (see Appendix B.3), using a prime-order group can potentially yield a more efficient algorithm $\mathcal{G}$ (which, recall, needs to compute a generator g of the group $\mathbb{G}$ that it outputs).

For some cryptographic constructions, the proof of security requires computing multiplicative inverses of certain exponents (we will see an example in Section 7.4.2). When the group order is a prime q, any non-zero exponent will be invertible modulo q, enabling this computation to be possible.

A final reason for working with prime-order groups applies in situations when the *decisional* Diffie-Hellman problem should be hard. Fixing a group $\mathbb{G}$ with generator g, the DDH problem boils down to distinguishing between tuples of the form $(h_1, h_2, \mathsf{DH}_g(h_1, h_2))$ for random h_1, h_2, and tuples of the form (h_1, h_2, y), for random h_1, h_2, y. A necessary condition for the DDH problem to be hard is that $\mathsf{DH}_g(h_1, h_2)$ by *itself* should be indistinguishable from a random group element. It seems that it would be best if $\mathsf{DH}_g(h_1, h_2)$ actually *were* a random group element when h_1 and h_2 are chosen at random.[3] We show that when the group order q is prime, this is (almost) true. In order to see this, we first prove the following:

PROPOSITION 7.61 *Let $\mathbb{G}$ be a group of prime order q with generator g. If x_1 and x_2 are chosen uniformly at random from $\mathbb{Z}_q$, then*

$$\Pr\left[\mathsf{DH}_g(g^{x_1}, g^{x_2}) = 1\right] = 1 - \left(1 - \frac{1}{q}\right)^2 = \frac{2}{q} - \frac{1}{q^2},$$

and for any other value $y \in \mathbb{G}$, $y \neq 1$:

$$\Pr[\mathsf{DH}_g(g^{x_1}, g^{x_2}) = y] = \frac{1}{q} \cdot \left(1 - \frac{1}{q}\right) = \frac{1}{q} - \frac{1}{q^2}.$$

PROOF We use the fact that $\mathsf{DH}_g(g^{x_1}, g^{x_2}) = g^{[x_1 \cdot x_2 \bmod q]}$. Since q is prime, $[x_1 \cdot x_2 \bmod q] = 0$ if and only if either $x_1 = 0$ or $x_2 = 0$. Because x_1 and x_2 are uniformly distributed in $\mathbb{Z}_q$,

$$\begin{aligned}\Pr[\mathsf{DH}_g(g^{x_1}, g^{x_2}) = 1] &= \Pr[x_1 = 0 \vee x_2 = 0] \\ &= 1 - \Pr[x_1 \neq 0] \cdot \Pr[x_2 \neq 0] = 1 - \left(1 - \frac{1}{q}\right)^2.\end{aligned}$$

[3] It is important to keep in mind the distinction between the distribution of $\mathsf{DH}_g(h_1, h_2)$, and the distribution of $\mathsf{DH}_g(h_1, h_2)$ *conditioned on the given values of* h_1, h_2. Since $\mathsf{DH}_g(h_1, h_2)$ is a deterministic function of h_1 and h_2, the latter distribution puts probability 1 on the correct answer $\mathsf{DH}_g(h_1, h_2)$ and is thus far from uniform. We are interested here in the distribution of $\mathsf{DH}_g(h_1, h_2)$ when h_1, h_2 are random and unknown.

Fix any $y \in \mathbb{G}$, $y \neq 1$, and let $x = \log_g y \neq 0$. Note that $\mathsf{DH}_g(g^{x_1}, g^{x_2}) = y$ if and only if $x_1 x_2 = x \bmod q$. Since q is prime, all non-zero elements of $\mathbb{Z}_q$ have a multiplicative inverse modulo q, and so $x_1 x_2 = x \bmod q$ if and only if x_1 is non-zero and $x_2 = x \cdot (x_1)^{-1} \bmod q$. So:

$$\begin{aligned}\Pr[\mathsf{DH}_g(g^{x_1}, g^{x_2}) = y] &= \Pr[x_1 x_2 = x \bmod q] \\ &= \Pr\left[x_2 = x \cdot (x_1)^{-1} \bmod q \mid x_1 \neq 0\right] \cdot \Pr[x_1 \neq 0] \\ &= \frac{1}{q} \cdot \left(1 - \frac{1}{q}\right),\end{aligned}$$

as claimed. ■

We now compare this to the uniform distribution over $\mathbb{G}$. A uniformly-distributed element y' has $\Pr[y' = y] = 1/q$ for all $y \in \mathbb{G}$ (i.e., including when $y = 1$). When $\|q\| = n$ (and so $q = \Theta(2^n)$) the above proposition says that for uniformly-distributed h_1 and h_2

$$\Pr[\mathsf{DH}_g(h_1, h_2) = y] = \frac{1}{q} \pm \mathsf{negl}(n).$$

In this sense, $\mathsf{DH}_g(h_1, h_2)$ is close to uniform in $\mathbb{G}$. The above notwithstanding, we stress that using a group of prime order is neither necessary nor sufficient for the DDH problem to be hard (indeed, the DDH problem is *easy* in the additive group $\mathbb{Z}_p$ for a prime p). Instead, it should merely be viewed as an additional, heuristic reason why prime-order groups are preferred.

7.3.3 Working in (Subgroups of) $\mathbb{Z}_p^*$

Groups of the form $\mathbb{Z}_p^*$, for a prime p, give one class of cyclic groups in which the discrete logarithm problem is believed to be hard. Concretely, let $\mathcal{G}_1$ be an algorithm that, on input 1^n, chooses a random n-bit prime p, and outputs p and the group order $q = p - 1$ along with a generator g of $\mathbb{Z}_p^*$. (Section 7.2.1 discusses efficient algorithms for choosing a random prime, and Appendix B.3 shows how to efficiently find a generator of $\mathbb{Z}_p^*$.) Then it is conjectured that the discrete logarithm problem is hard relative to $\mathcal{G}_1$.

The cyclic group $\mathbb{Z}_p^*$ (for $p > 3$ prime) does *not* have prime order. (The preference for groups of prime order was discussed in the previous section.) More problematic, the decisional Diffie-Hellman problem is *simply not hard in such groups* (see Exercise 11.10 of Chapter 11), and they are therefore unacceptable for the cryptographic applications we will explore in Chapters 9 and 10.

Thankfully, these problems can be addressed relatively easily by using an appropriate *subgroup* of $\mathbb{Z}_p^*$. Say an element $y \in \mathbb{Z}_p^*$ is a *quadratic residue modulo* p if there exists an $x \in \mathbb{Z}_p^*$ such that $x^2 = y \bmod p$. It is not hard to show that the set of quadratic residues modulo p forms a subgroup of $\mathbb{Z}_p^*$.

Moreover, when p is prime it can be shown that squaring modulo p is a two-to-one function, implying that exactly *half* the elements of $\mathbb{Z}_p^*$ are quadratic residues. See Section 11.1.1 for a proof of this fact as well as further discussion of quadratic residues modulo a prime p.

If p is a *strong* prime — i.e., $p = 2q + 1$ with q prime — then the subgroup of quadratic residues modulo p has exactly $(p-1)/2 = q$ elements. Since q is prime, Corollary 7.52 shows that this subgroup is cyclic and furthermore all elements of this subgroup (except the identity) are generators. For completeness, we sketch an appropriate polynomial-time algorithm $\mathcal{G}_2$ that follows easily from the above discussion.

ALGORITHM 7.62
A group generation algorithm $\mathcal{G}_2$

Input: Security parameter 1^n
Output: Cyclic group $\mathbb{G}$, its order q, and a generator g

generate a random $(n+1)$-bit strong prime p
$q := (p-1)/2$
choose an arbitrary $x \in \mathbb{Z}_p^*$ with $x \neq \pm 1 \bmod p$ and set $g := x^2 \bmod p$
return p, q, g

Note that p serves as a description of the group $\mathbb{G}$ of quadratic residues modulo p. The DDH problem is believed to be hard for $\mathcal{G}_2$ as above.

The strong prime p can be generated as described in Section 7.2.1. Note that g is computed in such a way that it is guaranteed to be a quadratic residue modulo p with $g \neq 1$, and so g will be a generator of the subgroup of quadratic residues modulo p.

7.3.4 * Elliptic Curve Groups

The groups we have seen thus far have all been based on modular arithmetic. Another interesting class of groups is those consisting of *points on elliptic curves*. Such groups are used in cryptographic applications since, in contrast to $\mathbb{Z}_p^*$, there is currently no known sub-exponential time algorithm for solving the discrete logarithm problem in such groups. (See Chapter 8 for further discussion.) Although elliptic curve groups are important in practical applications of cryptography, our treatment of such groups in this book is (unfortunately) scant for the following reasons:

1. The mathematics required for a deeper understanding of elliptic curve groups is more than we were willing to assume on the part of the reader. Our treatment of elliptic curves is therefore rather minimal and sacrifices generality in favor of simplicity. The reader interested in further exploring this topic is advised to consult the references at the end of the chapter.

2. Most cryptographic schemes based on elliptic-curve groups (and all the schemes in this book) can be analyzed and understood by treating the underlying group in a completely *generic* fashion, without reference to any particular group used to instantiate the scheme. For example, we will see in later chapters cryptographic schemes that can be based on *arbitrary* cyclic groups (possibly of prime order); these schemes are secure as long as some appropriate computational problem in the underlying group is "hard". From the perspective of provable security, then, it makes no difference how the group is actually instantiated (as long as the relevant computational problem is believed to be hard in the group). Of course, when it comes time to *implement* the scheme in practice, the concrete choice of which underlying group to use is of fundamental importance.

We now proceed with our brief treatment of elliptic curves. Let $p \geq 5$ be a prime.[4] Consider an equation E in the variables x and y of the form:

$$y^2 = x^3 + Ax + B \bmod p, \tag{7.1}$$

where $A, B \in \mathbb{Z}_p$ are constants with $4A^3 + 27B^2 \neq 0 \bmod p$ (this latter condition ensures that the equation $x^3 + Ax + B = 0 \bmod p$ has no repeated roots). Let $\hat{E}(\mathbb{Z}_p)$ denote the set of pairs $(x, y) \in \mathbb{Z}_p \times \mathbb{Z}_p$ satisfying the above equation; i.e.,

$$\hat{E}(\mathbb{Z}_p) \stackrel{\text{def}}{=} \left\{(x, y) \mid x, y \in \mathbb{Z}_p \text{ and } y^2 = x^3 + Ax + B \bmod p\right\}.$$

Define $E(\mathbb{Z}_p) \stackrel{\text{def}}{=} \hat{E}(\mathbb{Z}_p) \cup \{\mathcal{O}\}$, where $\mathcal{O}$ is a special value whose purpose we will discuss shortly. The elements of the set $E(\mathbb{Z}_p)$ are called the *points* on the *elliptic curve* E defined by Equation (7.1), and $\mathcal{O}$ is called the "point at infinity."

Example 7.63

Recall that an element $y \in \mathbb{Z}_p^*$ is a *quadratic residue modulo* p if there exists an $x \in \mathbb{Z}_p^*$ such that $x^2 = y \bmod p$; we say that x is a *square root of* y in this case. Furthermore, when $p > 2$ is prime, every quadratic residue modulo p has exactly two square roots. (See Section 11.1.1 for further discussion.)

Let $f(x) \stackrel{\text{def}}{=} x^3 + 3x + 3$ and consider the curve $E : y^2 = f(x) \bmod 7$. Each value of x for which $f(x)$ is a quadratic residue modulo 7 yields two points on the curve, values x for which $f(x)$ is a non-quadratic residue are not on the

[4] The theory can be adapted to deal with the case of $p = 2$ or 3 but this introduces additional complications. For the advanced reader, we mention that elliptic curves can in fact be defined over arbitrary (finite or infinite) *fields*, and the discussion here carries over to fields of characteristic not equal to 2 or 3.

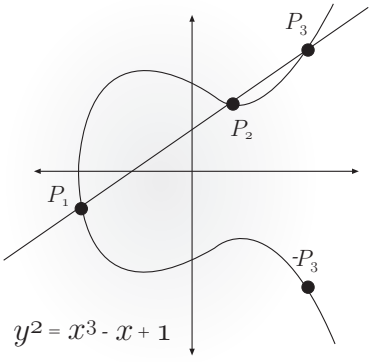

FIGURE 7.2: An elliptic curve over the reals.

curve, and values of x for which $f(x) = 0 \bmod 7$ give one point on the curve. This allows us to determine the points on the curve:

- $f(0) = 3 \bmod 7$, a quadratic non-residue modulo 7.
- $f(1) = 0 \bmod 7$, so we obtain the point $(1, 0) \in E(\mathbb{Z}_7)$.
- $f(2) = 3 \bmod 7$, a quadratic non-residue modulo 7.
- $f(3) = 4 \bmod 7$, a quadratic residue modulo 7 with square roots 2 and 5. This yields the points $(3, 2), (3, 5) \in E(\mathbb{Z}_7)$.
- $f(4) = 2 \bmod 7$, a quadratic residue modulo 7 with square roots 3 and 4. This yields the points $(4, 3), (4, 4) \in E(\mathbb{Z}_7)$.
- $f(5) = 3 \bmod 7$, a quadratic non-residue modulo 7.
- $f(6) = 6 \bmod 7$, a quadratic non-residue modulo 7.

Including the point at infinity, there are 6 points in $E(\mathbb{Z}_7)$. ♢

A useful way to conceptualize $E(\mathbb{Z}_p)$ is to look at the graph of Equation (7.1) over the reals (i.e., the equation $y^2 = x^3 + Ax + B$ without reduction modulo p) as in Figure 7.2. This figure does not correspond exactly to $E(\mathbb{Z}_p)$ because, for example, $E(\mathbb{Z}_p)$ has a finite number of points ($\mathbb{Z}_p$ is, after all, a finite set) while there are an infinite number of solutions to the same equation if we allow x and y to range over all real numbers. Nevertheless, the picture provides useful intuition. In such a figure, one can think of the "point at infinity" $\mathcal{O}$ as sitting at the top of the y-axis and lying on every vertical line.

It can be shown that every line intersecting the curve E intersects the curve in exactly 3 points, where: (1) a point P is counted twice if the line is tangent to the curve at P, and (2) the point at infinity is also counted (when the line

is vertical). This fact is used to define a binary operation, called 'addition' and denoted by '+,' on points of $E(\mathbb{Z}_p)$ in the following way:[5]

- The point $\mathcal{O}$ is defined as an (additive) identity; that is, for all $P \in E(\mathbb{Z}_p)$ we define $P + \mathcal{O} = \mathcal{O} + P = P$.
- If P_1, P_2, P_3 are co-linear points on E then we require that

$$P_1 + P_2 + P_3 = \mathcal{O}. \tag{7.2}$$

  (This disregards the ordering of P_1, P_2, P_3, implying that addition is commutative for all points, and associative for co-linear points.)

Rules for negation and addition of arbitrary points follow from the above.

Negation. Given a point P, the negation $-P$ is (by definition of negation) that point for which $P+(-P) = \mathcal{O}$. If $P = \mathcal{O}$ then $-P = \mathcal{O}$. Otherwise, since $P + (-P) + \mathcal{O} = (P + (-P)) + \mathcal{O} = \mathcal{O} + \mathcal{O} = \mathcal{O}$ we see, using Equation (7.2), that $-P$ corresponds to the third point on the line passing through P and $\mathcal{O}$ or, equivalently, the vertical line passing through P. As can be seen by looking at Figure 7.2, this means that $-P$ is simply the reflection of P in the x-axis; that is, if $P = (x, y)$ then $-P = (x, -y)$.

Addition of points. For two arbitrary points $P_1, P_2 \neq \mathcal{O}$ on E, we can evaluate their sum P_1+P_2 by drawing the line through P_1, P_2 (if $P_1 = P_2$ then draw the line tangent to E at P_1) and finding the third point of intersection P_3 of this line with E; the third point of intersection may be $P_3 = \mathcal{O}$ if the line is vertical. Equation (7.2) implies that $P_1 + P_2 + P_3 = \mathcal{O}$, or $P_1 + P_2 = -P_3$. If $P_3 = \mathcal{O}$ then $P_1 + P_2 = -\mathcal{O} = \mathcal{O}$. Otherwise, if the third point of intersection of the line through P_1 and P_2 is the point $P_3 = (x, y) \neq \mathcal{O}$ then

$$P_1 + P_2 = -P_3 = (x, -y).$$

Graphically, $P_1 + P_2$ can be found by finding the third point of intersection of E and the line through P_1 and P_2, and then reflecting in the x-axis.

It is straightforward, but tedious, to work out the addition law concretely. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points in $E(\mathbb{Z}_p)$, with $P_1, P_2 \neq \mathcal{O}$ and E as in Equation (7.1). To keep matters simple, suppose $x_1 \neq x_2$ (the extension to the case $x_1 = x_2$ is still straightforward but even more tedious). The slope of the line through these points is

$$m = \frac{y_2 - y_1}{x_2 - x_1} \bmod p\,;$$

our assumption that $x_1 \neq x_2$ means that $x_2 - x_1 \neq 0 \bmod p$ and so the inverse of $(x_2 - x_1)$ modulo p exists. The line passing through P_1 and P_2 has the equation

$$y = m \cdot (x - x_1) + y_1 \bmod p\,.$$

[5] Our approach is informal, and we do not justify that it leads to a consistent definition.

To find the third point of intersection of this line with E, substitute the above into the equation for E to obtain

$$\Big(m \cdot (x - x_1) + y_1\Big)^2 = x^3 + Ax + B \bmod p.$$

The values of x that satisfy this equation are x_1, x_2, and $[m^2 - x_1 - x_2 \bmod p]$. The first two solutions correspond to the original points P_1 and P_2, while the third is the x-coordinate of the third point of intersection P_3. The y-value corresponding to this third value of x is $y = [m \cdot (x - x_1) + y_1 \bmod p]$. That is, $P_3 = (x_3, y_3)$ where

$$x_3 = [m^2 - x_1 - x_2 \bmod p] \quad \text{and} \quad y_3 = [m \cdot (x_3 - x_1) + y_1 \bmod p].$$

To obtain the desired answer $P_1 + P_2$, it remains only to take the negation of P_3 (or, equivalently, reflect P_3 in the x-axis) giving:

PROPOSITION 7.64 *Let $p \geq 5$ be prime, and $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on the elliptic curve $y^2 = x^3 + Ax + B \bmod p$ with $P_1, P_2 \neq \mathcal{O}$ and $x_1 \neq x_2$. Then $P_1 + P_2 = (x_3, y_3)$ with*

$$x_3 = [m^2 - x_1 - x_2 \bmod p] \quad \text{and} \quad y_3 = [m \cdot (x_1 - x_3) - y_1 \bmod p],$$

where $m = \left[\frac{y_2 - y_1}{x_2 - x_1} \bmod p\right]$.

For completeness, we state the addition law for points not covered by the above proposition.

PROPOSITION 7.65 *Let $p \geq 5$ be prime, and $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on the elliptic curve $y^2 = x^3 + Ax + B \bmod p$ with $P_1, P_2 \neq \mathcal{O}$.*

1. *If $x_1 = x_2$ but $y_1 \neq y_2$ then $P_1 = -P_2$ and so $P_1 + P_2 = \mathcal{O}$.*
2. *If $P_1 = P_2$ and $y_1 = 0$ then $P_1 + P_2 = 2P_1 = \mathcal{O}$.*
3. *If $P_1 = P_2$ and $y_1 \neq 0$ then $P_1 + P_2 = (x_3, y_3)$ with*

$$x_3 = [m^2 - 2x_1 \bmod p] \quad \text{and} \quad y_3 = [m \cdot (x_1 - x_3) - y_1 \bmod p],$$

where $m = \left[\frac{3x_1^2 + A}{2y_1} \bmod p\right]$.

Somewhat amazingly, it can be shown the set of points $E(\mathbb{Z}_p)$ along with the addition rule defined above form an abelian group! Actually, we have already seen almost all the necessary properties: closure under addition follows from the fact (not proven here) that any line intersecting E has three points of intersection; $\mathcal{O}$ acts as the identity; each point on $E(\mathbb{Z}_p)$ has an inverse in $E(\mathbb{Z}_p)$; and commutativity of addition follows from Equation (7.2). The

difficult property to verify is associativity, which the disbelieving reader can check through tedious calculation. A more illuminating proof that does not involve explicit calculation relies on algebraic geometry.

In typical cryptographic applications, parameters of the elliptic curve are chosen in such a way that the group $E(\mathbb{Z}_p)$ (or a subgroup thereof) is of prime-order, and hence a cyclic group. Efficient methods for doing so are beyond the scope of this book.

7.4 Cryptographic Applications of Number-Theoretic Assumptions

We have spent a fair bit of time discussing number theory and group theory, and introducing computational hardness assumptions that are widely believed to hold. Applications of these assumptions will occupy us for the rest of the book, but we provide some brief examples here.

7.4.1 One-Way Functions and Permutations

One-way functions are the minimal cryptographic primitive, and they are both necessary and sufficient for all the private-key constructions we have seen in Chapters 3 and 4. A more complete discussion of the role of one-way functions in cryptography is given in Chapter 6; here we only provide a definition of one-way functions and demonstrate that their existence follows from all the number-theoretic hardness assumptions we have seen in this chapter.

Informally, a function f is *one-way* if it is easy to compute but hard to invert. The following experiment and definition is a formal statement of this (and is a re-statement of Definition 6.1):

The inverting experiment $\mathsf{Invert}_{\mathcal{A},f}(n)$**:**

1. *Choose input* $x \leftarrow \{0,1\}^n$*. Compute* $y := f(x)$*.*
2. $\mathcal{A}$ *is given* 1^n *and* y *as input, and outputs* x'*.*
3. *The output of the experiment is defined to be 1 if and only if* $f(x') = y$*.*

DEFINITION 7.66 *A function* $f : \{0,1\}^* \to \{0,1\}^*$ *is* one-way *if the following two conditions hold:*

1. (Easy to compute:) *There exists a polynomial-time algorithm that on input* x *outputs* $f(x)$*.*
2. (Hard to invert:) *For all probabilistic polynomial-time algorithms* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1] \leq \mathsf{negl}(n).$$

We now show formally that the factoring assumption implies the existence of a one-way function. Let Gen be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$ and p and q are n-bit primes except with probability negligible in n. (We use Gen rather than GenModulus here purely for notational convenience.) Since Gen runs in polynomial time, there exists a polynomial p such that the number of random bits the algorithm uses on input 1^n is at most $p(n)$. For simplicity, assume that Gen always uses *exactly* $p(n)$ bits on input 1^n, and further that $p(n)$ is strictly increasing. In Algorithm 7.67 we define a function f_{Gen} that uses its input as a random tape for Gen. Since the hardness of inverting a one-way function is only required for random inputs, we are able to interpret the input as a random tape and the generation algorithm will have the required properties. Thus, although the algorithm for computing f_{Gen} runs Gen as a subroutine, it is actually *deterministic*, as required. (The computation of n based on the length of the input x is a technicality that is needed to make sure that Gen receives a random tape of the appropriate length.)

ALGORITHM 7.67
Algorithm computing f_{Gen}

Input: String x
Output: String N

compute n such that $p(n) \leq |x| < p(n+1)$
compute $(N, p, q) := \mathsf{Gen}(1^n; x)$
  /* i.e., run $\mathsf{Gen}(1^n)$ using x as the random tape */
return N

If the factoring problem is hard relative to Gen then, intuitively, f_{Gen} is a one-way function. Certainly f_{Gen} is easy to compute. As for the hardness of inverting this function, for any n' the following distributions are identical:

1. The modulus N output by $f_{\mathsf{Gen}}(x)$, when $x \in \{0,1\}^{n'}$ is chosen according to the uniform distribution.

2. The modulus N output by the randomized process in which $\mathsf{Gen}(1^n)$ is run to obtain N. Here, n satisfies $p(n) \leq n' < p(n+1)$.

Since moduli N chosen according to the second distribution are hard to factor, the same holds for moduli N chosen according to the first distribution. Moreover, given any x for which $f_{\mathsf{Gen}}(x) = N$, it is easy to recover a factor of N (by running $\mathsf{Gen}(1^n; x)$ to obtain (N, p, q) and outputting the factors p and q). Thus, inverting f_{Gen} and finding such an x is as hard as factoring. We therefore have the following theorem (a formal proof follows fairly easily from what we have said):

THEOREM 7.68 *If the factoring problem is hard relative to* Gen*, then* f_{Gen} *is a one-way function.*

A corollary is that hardness of the RSA problem implies the existence of a one-way function (this follows from the fact that hardness of RSA implies that factoring is hard, at least when GenRSA is constructed as in Algorithm 7.47).

* One-Way Permutations

We have seen that one-way functions exist if the RSA problem is hard. However, the RSA problem actually gives us something much stronger: a family of one-way *permutations*. (The material from here until the end of this section is needed for Section 10.7 and will be meaningful to those who have studied Chapter 6; otherwise, it may be skipped.) We begin with a re-statement of Definitions 6.3 and 6.4:

DEFINITION 7.69 *A tuple* $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ *of probabilistic polynomial-time algorithms is a* family of functions *if the following hold:*

1. *The* parameter generation algorithm Gen*, on input* 1^n*, outputs parameters* I *with* $|I| \geq n$*. Each value of* I *output by* Gen *defines sets* $\mathcal{D}_I$ *and* $\mathcal{R}_I$ *that constitute the domain and range, respectively, of a function* f_I *defined below.*
2. *The* sampling algorithm Samp*, on input* I*, outputs a uniformly distributed element of* $\mathcal{D}_I$ *(except possibly with probability negligible in* $|I|$*).*
3. *The deterministic* evaluation algorithm f*, on input* I *and* $x \in \mathcal{D}_I$*, outputs an element* $y \in \mathcal{R}_I$*. We write this as* $y := f_I(x)$*.*

Π *is a* family of permutations *if, for each value of* I *output by* $\mathsf{Gen}(1^n)$*, it holds that* $\mathcal{D}_I = \mathcal{R}_I$ *and the function* $f_I : \mathcal{D}_I \to \mathcal{D}_I$ *is a bijection.*

Due to the last condition, when Π is a family of permutations, choosing $x \leftarrow \mathcal{D}_I$ uniformly at random and setting $y := f_I(x)$ results in a value of y that is uniformly distributed in $\mathcal{D}_I$.

Given a family of functions Π, consider the following experiment for any algorithm $\mathcal{A}$ and parameter n:

The inverting experiment $\mathsf{Invert}_{\mathcal{A},\Pi}(n)$**:**

1. $\mathsf{Gen}(1^n)$ *is run to obtain* I*, and then* $\mathsf{Samp}(I)$ *is run to obtain a random* $x \leftarrow \mathcal{D}_I$*. Finally,* $y := f_I(x)$ *is computed.*
2. $\mathcal{A}$ *is given* I *and* y *as input, and outputs* x'*.*
3. *The output of the experiment is defined to be 1 if* $f_I(x') = y$*, and 0 otherwise.*

DEFINITION 7.70 *A family of functions/permutations* $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ *is* one-way *if for all probabilistic polynomial-time algorithms* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Invert}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

Given GenRSA as in Section 7.2.4, Construction 7.71 defines a family of permutations. It is immediate that if the RSA problem is hard for GenRSA then this family is in fact *one-way*. It can similarly be shown that hardness of the discrete logarithm problem in $\mathbb{Z}_p^*$, with p prime, implies the existence of a one-way permutation family; see Exercise 7.20.

CONSTRUCTION 7.71

Let GenRSA be as in Section 7.2.4. Define a family of permutations as follows:

- Gen: on input 1^n, run $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and output $I = \langle N, e \rangle$. Set $\mathcal{D}_I = \mathbb{Z}_N^*$.
- Samp: on input $I = \langle N, e \rangle$, choose a random element of $\mathbb{Z}_N^*$.
- f: on input $I = \langle N, e \rangle$ and $x \in \mathbb{Z}_N^*$, output $[x^e \bmod N]$.

A family of one-way permutations (assuming the RSA problem is hard relative to GenRSA).

7.4.2 Constructing Collision-Resistant Hash Functions

Collision-resistant hash functions were introduced in Section 4.6. Although, as discussed in that section, there exist heuristic constructions of collision-resistant hash functions that are used widely in practice, we have not yet seen any constructions of such hash functions that can be proven secure under more basic assumptions. (In particular, no such constructions were shown in Chapter 6. In fact, there is evidence that constructing collision-resistant hash functions from arbitrary one-way functions or permutations is *impossible*.)

We show now a construction of a collision-resistant hash function based on the discrete logarithm assumption in prime-order groups. A second construction based on the RSA problem is described in Exercise 7.21. Although these constructions are less efficient than the hash functions used in practice, they are important since they illustrate the *feasibility* of achieving collision resistance based on standard and well-studied cryptographic assumptions.

As in Section 7.3.2, let $\mathcal{G}$ be a polynomial-time algorithm that, on input 1^n, outputs a (description of a) cyclic group $\mathbb{G}$, its order q (with $\|q\| = n$), and a generator g. As always, we assume that the group operation in $\mathbb{G}$ can be computed efficiently. Finally, we also require that q is *prime* except possibly with negligible probability. (Recall that there is a general preference for using

groups of prime order, as discussed in Section 7.3.2.) A fixed-length hash function based on $\mathcal{G}$ is given in Construction 7.72.

CONSTRUCTION 7.72

Let $\mathcal{G}$ be as described in the text. Define a fixed-length hash function (Gen, H) as follows:

- Gen: on input 1^n, run $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$ and then select $h \leftarrow \mathbb{G}$. Output $s := \langle \mathbb{G}, q, g, h \rangle$ as the key.
- H: given a key $s = \langle \mathbb{G}, q, g, h \rangle$ and input $(x_1, x_2) \in \mathbb{Z}_q \times \mathbb{Z}_q$, output $H^s(x_1, x_2) := g^{x_1} h^{x_2}$.

A fixed-length hash function.

Note that Gen and H can be computed in polynomial time. Before continuing with an analysis of the construction, we make some technical remarks:

- For a given $s = \langle \mathbb{G}, q, g, h \rangle$ with $n = \|q\|$, the function H^s is described as taking elements of $\mathbb{Z}_q \times \mathbb{Z}_q$ as input. However, H^s can be viewed as taking bit-strings of length $2 \cdot (n-1)$ as input if we parse inputs $x \in \{0,1\}^{2(n-1)}$ as two strings x_1, x_2 each of length $n-1$, and then view each of x_1, x_2 as an element of $\mathbb{Z}_q$ in the natural way.
- The output of H^s is similarly specified as being an element of $\mathbb{G}$, but we can view this as a bit-string if we fix some representation of $\mathbb{G}$. To satisfy the requirements of Definition 4.12 (which requires the output length to be fixed as a function of n) we can pad the output as needed.
- Given the above, the construction only compresses its input for certain groups $\mathbb{G}$ (specifically, when elements of $\mathbb{G}$ can be represented using fewer than $2n-2$ bits). As shown in Exercise 7.19, compression can be achieved when using subgroups of $\mathbb{Z}_p^*$ of the type discussed in Section 7.3.3. A generalization of Construction 7.72 can be used to obtain compression from *any* $\mathcal{G}$ for which the discrete logarithm problem is hard, regardless of the number of bits required to represent group elements; see Exercise 7.22.

THEOREM 7.73 *If the discrete logarithm problem is hard relative to $\mathcal{G}$, then Construction 7.72 is a fixed-length collision-resistant hash function (subject to the discussion regarding compression, above).*

PROOF Let $\Pi = (\mathsf{Gen}, H)$ as in Construction 7.72, and let $\mathcal{A}$ be a probabilistic polynomial-time algorithm with

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n) = 1]$$

(cf. Definition 4.12). We show how $\mathcal{A}$ can be used by an algorithm $\mathcal{A}'$ to solve the discrete logarithm problem with success probability $\varepsilon(n)$:

Algorithm $\mathcal{A}'$:
The algorithm is given $\mathbb{G}, q, g, h$ as input.

1. Let $s := \langle \mathbb{G}, q, g, h \rangle$. Run $\mathcal{A}(s)$ and obtain output x and x'.
2. If $x \neq x'$ and $H^s(x) = H^s(x')$ then:
   (a) If $h = 1$ return 0
   (b) Otherwise ($h \neq 1$), parse x as (x_1, x_2) and parse x' as (x'_1, x'_2). Return $\left[(x_1 - x'_1) \cdot (x'_2 - x_2)^{-1} \bmod q\right]$.

Clearly, $\mathcal{A}'$ runs in polynomial time. Furthermore, the input s given to $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is distributed exactly as in experiment $\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}$ for the same value of the security parameter n. (The input to $\mathcal{A}'$ is generated by running $\mathcal{G}(1^n)$ to obtain $\mathbb{G}, q, g$ and then choosing $h \in \mathbb{G}$ uniformly at random. This is exactly how s is generated by $\mathsf{Gen}(1^n)$.) So, with probability exactly $\varepsilon(n)$ there is a *collision*: i.e., $x \neq x'$ and $H^s(x) = H^s(x')$.

We claim that whenever there is a collision, $\mathcal{A}'$ returns the correct answer $\log_g h$. If $h = 1$ then this is clearly true (since $\log_g h = 0$ in this case). Otherwise, the existence of a collision means that

$$\begin{aligned} H^s(x_1, x_2) = H^s(x'_1, x'_2) &\Rightarrow g^{x_1} h^{x_2} = g^{x'_1} h^{x'_2} \\ &\Rightarrow g^{x_1 - x'_1} = h^{x'_2 - x_2}. \end{aligned} \tag{7.3}$$

Let $\Delta \stackrel{\text{def}}{=} x'_2 - x_2$. Note that $\Delta \neq 0 \bmod q$ since this would imply that $[(x_1 - x'_1) \bmod q] = 0$, but then $x = (x_1, x_2) = (x'_1, x'_2) = x'$ in contradiction to the assumption that $x \neq x'$. Since q is prime and $\Delta \neq 0 \bmod q$, the inverse $[\Delta^{-1} \bmod q]$ exists. Raising each side of Equation (7.3) to this power gives:

$$g^{(x_1 - x'_1) \cdot \Delta^{-1}} = \left(h^{x'_2 - x_2}\right)^{[\Delta^{-1} \bmod q]} = h^{[\Delta \cdot \Delta^{-1} \bmod q]} = h^1 = h,$$

and so

$$\log_g h = [(x_1 - x'_1)\Delta^{-1} \bmod q] = \left[(x_1 - x'_1) \cdot (x'_2 - x_2)^{-1} \bmod q\right],$$

the output returned by $\mathcal{A}'$.

We see that $\mathcal{A}'$ correctly solves the discrete logarithm problem with probability exactly $\varepsilon(n)$. Since, by assumption, the discrete logarithm problem is hard relative to $\mathcal{G}$, we conclude that $\varepsilon(n)$ is negligible. ∎

Using Exercise 7.22 in combination with the Merkle-Damgård transform (see Section 4.6.4), we have the following theorem:

THEOREM 7.74 *If there exists a probabilistic polynomial-time algorithm $\mathcal{G}$ relative to which the discrete logarithm problem is hard, then there exists a collision resistant hash function.*

References and Additional Reading

The book by Childs [32] has excellent coverage of the group theory discussed in this chapter (and more), in greater depth but at a similar level of exposition. Shoup [131] gives a more advanced, yet still accessible, treatment of much of this material also. Relatively gentle introductions to abstract algebra and group theory that go well beyond what we have space for here are available in the books by Fraleigh [57] and Herstein [76]; the interested reader will have no trouble finding more advanced algebra texts if they are so inclined.

The first efficient primality test was by Solovay and Strassen [135]. The Miller-Rabin test is due to Miller [105] and Rabin [120]. A deterministic primality test was recently discovered by Agrawal et al. [2]. See Dietzfelbinger [46] for an accessible, comprehensive presentation of primality testing.

The RSA permutation was introduced by Rivest, Shamir, and Adleman [122]. Boneh [25] summarizes the status of this problem as of 1999. The discrete logarithm and Diffie-Hellman problems were first considered, at least implicitly, by Diffie and Hellman [47]. Recent surveys of each of these problems and their applications are given by Odlyzko [112] and Boneh [24].

While there are many references to elliptic curves, there are almost none that do not require an advanced mathematical background on the part of the reader. The book by Silverman and Tate [132] is perhaps an exception. As with most books on the subject, however, that book has little coverage of elliptic curves over *finite* fields, which is the case most relevant to cryptography. The text by Washington [145], though a bit more advanced, deals heavily (though not exclusively) with the finite-field case. Implementation issues related to elliptic-curve cryptography are covered by Hankerson et al. [72].

The construction of a collision-resistant hash function based on the discrete logarithm problem is due to [31], and an earlier construction based on the hardness of factoring is given in [70] (see also Exercise 7.21).

Exercises

7.1 Let $\mathbb{G}$ be an abelian group. Prove that there is a *unique* identity in $\mathbb{G}$, and that every element $g \in \mathbb{G}$ has a *unique* inverse.

7.2 Show that Proposition 7.36 does not necessarily hold when $\mathbb{G}$ is infinite.

Hint: Consider the set $\{1\} \cup \{2, 4, 6, 8, \ldots\} \subset \mathbb{R}$.

7.3 Let $\mathbb{G}$ be a finite group, and $g \in \mathbb{G}$. Show that $\langle g \rangle$ is a subgroup of $\mathbb{G}$. Is the set $\{g^0, g^1, \ldots\}$ necessarily a subgroup of $\mathbb{G}$ when $\mathbb{G}$ is infinite?

7.4 This question concerns the Euler phi function.

(a) Let p be a prime and $e \geq 1$ an integer. Show that

$$\phi(p^e) = p^{e-1}(p-1).$$

(b) Let p, q be relatively prime. Show that $\phi(pq) = \phi(p) \cdot \phi(q)$. (You may not use the Chinese remainder theorem.)

(c) Prove Theorem 7.19.

7.5 Compute the final two (decimal) digits of 3^{1000} (by hand).

Hint: The answer is $[3^{1000} \bmod 100]$.

7.6 Compute $[101^{4,800,000,023} \bmod 35]$ (by hand).

7.7 Prove that if $\mathbb{G}, \mathbb{H}$ are groups, then $\mathbb{G} \times \mathbb{H}$ is a group.

7.8 Let p, N be integers with $p \,|\, N$. Prove that for any integer X,

$$[[X \bmod N] \bmod p\,] = [X \bmod p\,].$$

Show that, in contrast, $[[X \bmod p\,] \bmod N]$ need not equal $[X \bmod N]$.

7.9 Complete the details of the proof of the Chinese remainder theorem, showing that $\mathbb{Z}_N^*$ is isomorphic to $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.

7.10 Corollary 7.21 shows that if $N = pq$ and $ed = 1 \bmod \phi(N)$ then for all $x \in \mathbb{Z}_N^*$ we have $(x^e)^d = x \bmod N$. Show that this holds for all $x \in \mathbb{Z}_N$.

Hint: Use the Chinese remainder theorem.

7.11 This exercise develops an efficient algorithm for testing whether an integer is a perfect power.

(a) Show that if $N = \hat{N}^e$ for some integers $\hat{N}, e > 1$ then $e \leq \|N\| + 1$.

(b) Given N and e with $2 \leq e \leq \|N\| + 1$, show how to determine in $\mathsf{poly}(\|N\|)$ time whether there exists an integer $\hat{N}$ with $\hat{N}^e = N$.

Hint: Use binary search.

(c) Given N, show how to test in $\mathsf{poly}(\|N\|)$ time whether N is a perfect power.

7.12 Given N and $a \in \mathbb{Z}_N^*$, show how to test in polynomial time whether a is a strong witness that N is composite.

7.13 Let $N = pq$ be a product of two distinct primes. Show that if $\phi(N)$ and N are known, then it is possible to compute p and q in polynomial time.

Hint: Derive a quadratic equation (over the integers) in the unknown p.

7.14 Let $N = pq$ be a product of two distinct primes. Show that if N and an integer d such that $3 \cdot d = 1 \bmod \phi(N)$ are known, then it is possible to compute p and q in polynomial time.

Hint: Obtain a small list of possibilities for $\phi(N)$ and then use the previous exercise.

7.15 Prove formally that the hardness of the CDH problem relative to $\mathcal{G}$ implies the hardness of the discrete logarithm problem relative to $\mathcal{G}$.

7.16 Prove formally that the hardness of the DDH problem relative to $\mathcal{G}$ implies the hardness of the CDH problem relative to $\mathcal{G}$.

7.17 Prove the third statement in Proposition 7.65.

7.18 Determine whether or not the following problem is hard. Let p be prime, and fix $x \in \mathbb{Z}_{p-1}^*$. Given p, x, and $y := [g^x \bmod p]$ (where g is a random value between 1 and $p - 1$), find g; i.e., compute $y^{1/x} \bmod p$. If you claim the problem is hard, show a reduction to one of the assumptions introduced in this chapter. If you claim the problem is easy, present an algorithm, justify its correctness, and analyze its complexity.

7.19 Let $\mathcal{G}$ be an algorithm that, on input 1^n, outputs p, q, g where $p = 2q+1$ is a strong prime and g is a generator of the subgroup of quadratic residues modulo p. (See Section 7.3.3.) Show how to obtain compression in Construction 7.72 for p large enough.

7.20 Let $\mathcal{G}_1$ be as in Section 7.3.3. Show that hardness of the discrete logarithm problem relative to $\mathcal{G}_1$ implies the existence of a family of one-way permutations.

Hint: Define a permutation on elements of $\mathbb{Z}_p^*$.

7.21 Let GenRSA be as in Section 7.2.4. Prove that if the RSA problem is hard relative to GenRSA then Construction 7.75 shown below is a fixed-length collision-resistant hash function.

CONSTRUCTION 7.75

Define (Gen, H) as follows:

- Gen: on input 1^n, run $\mathsf{GenRSA}(1^n)$ to obtain N, e, d, and select $y \leftarrow \mathbb{Z}_N^*$. The key is $s := \langle N, e, y \rangle$.
- H: if $s = \langle N, e, y \rangle$, then H^s maps inputs in $\{0,1\}^{3n}$ to outputs in $\mathbb{Z}_N^*$. Let $f_0^s(x) \stackrel{\text{def}}{=} [x^e \bmod N]$ and $f_1^s(x) \stackrel{\text{def}}{=} [y \cdot x^e \bmod N]$. For a $3n$-bit long string $x = x_1 \cdots x_{3n}$, define

$$H^s(x) \stackrel{\text{def}}{=} f_{x_1}^s\left(f_{x_2}^s\left(\cdots\left(1\right)\cdots\right)\right).$$

Hint: Show that the eth root of y can be computed from any collision.

7.22 Consider the generalization of Construction 7.72 shown below.

CONSTRUCTION 7.76

Define a fixed-length hash function (Gen, H) as follows:

- Gen: on input 1^n, run $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, h_1)$ and then select $h_2, \ldots, h_t \leftarrow \mathbb{G}$. Output $s := \langle \mathbb{G}, q, (h_1, \ldots, h_t) \rangle$ as the key.
- H: given a key $s = \langle \mathbb{G}, q, (h_1, \ldots, h_t) \rangle$ and input $(x_1, \ldots, x_t)$ with $x_i \in \mathbb{Z}_q$, output $H^s(x_1, \ldots, x_t) := \prod_i h_i^{x_i}$.

- Prove that if the discrete logarithm problem is hard relative to $\mathcal{G}$, and q is prime, then the construction is a fixed-length collision-resistant hash function for any $t = \mathsf{poly}(n)$.
- Discuss how this construction can be used to obtain *compression* regardless of the number of bits needed to represent elements of $\mathbb{G}$ (as long as it is polynomial in n).

Chapter 8

* Algorithms for Factoring and Computing Discrete Logarithms

As discussed in Chapter 7, there are currently no known *polynomial-time* algorithms for factoring or for computing discrete logarithms in certain groups. But this does not mean that brute-force search is the best available approach for attacking these problems! Here, we survey some more efficient algorithms for these problems. These algorithms are interesting in their own right, and serve as a nice application of some of the number theory we have already learned. Moreover, understanding the effectiveness of these algorithms is crucial for choosing cryptographic parameters in practice. If a cryptographic scheme based on factoring is supposed to withstand adversaries mounting a dedicated attack for 15 years, then — at a minimum! — the modulus N used by the scheme needs to be long enough so that the best-known factoring algorithm will take (at least) 15 years to successfully factor N.

Algorithms for other problems. We focus here on algorithms for factoring and computing discrete logarithms, not on algorithms for, say, solving the RSA or decisional Diffie-Hellman problems. This may seem somewhat misguided since the latter are used much more often than the former when constructing cryptographic schemes and, as noted in the previous chapter, the latter are potentially easier to solve than the former. (That is, solving the RSA problem is potentially easier than factoring, and solving the decisional Diffie-Hellman problem is possibly easier than computing discrete logarithms.) Our focus is justified by the fact that, currently, the best known algorithm for solving RSA is to first factor the modulus, and (in appropriate groups as discussed in Sections 7.3.3 and 7.3.4) the best known algorithm for solving the decisional Diffie-Hellman problem is to compute discrete logarithms.

8.1 Algorithms for Factoring

Throughout this section, we assume that $N = pq$ is a product of two distinct primes. We will also be most interested in the case that p and q are each of the same (known) length n, and so $n = \Theta(\log N)$. There exist other factoring

algorithms tailored to work for N of a different form (e.g., when $N = p^r q$ for p, q prime and an integer $r > 1$, or when p and q have significantly different lengths) but we do not cover these here.

We will frequently use the Chinese remainder theorem along with the notation developed in Section 7.1.5. The Chinese remainder theorem states that

$$\mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*,$$

with isomorphism given by $f(x) \stackrel{\text{def}}{=} ([x \bmod p\,],\ [x \bmod q\,])$ for $x \in \mathbb{Z}_N^*$. The fact that f is an isomorphism means in particular that it gives a bijection between elements $x \in \mathbb{Z}_N^*$ and pairs $(x_p, x_q) \in \mathbb{Z}_p^* \times \mathbb{Z}_q^*$. We write $x \leftrightarrow (x_p, x_q)$, with $x_p = [x \bmod p\,]$ and $x_q = [x \bmod q\,]$, to denote this bijection.

Recall from Section 7.2 that *trial division*, a trivial, brute-force factoring method, finds a factor of a given number N with probability 1 in time $\mathcal{O}(\sqrt{N} \cdot \mathsf{polylog}(N))$. A more sophisticated factoring algorithm is therefore only interesting if its running time is asymptotically less than this. We cover three different factoring algorithms with improved running time:

- *Pollard's $p-1$ method* is effective when $p-1$ has "small" prime factors.
- *Pollard's rho method* applies to arbitrary N. (As such, it is called a *general-purpose* factoring algorithm.) Its running time for N of the form discussed at the beginning of this section is $\mathcal{O}(N^{1/4} \cdot \mathsf{polylog}(N))$. Since $N = 2^{\Theta(n)}$ this is still *exponential* in n, the length of N.
- The *quadratic sieve algorithm* is a general-purpose factoring algorithm that runs in time *sub-exponential* in the length of N.[1] We give a high-level overview of how this algorithm works, but the details are somewhat complex and are beyond the scope of this book.

Currently, the best-known general-purpose factoring algorithm (in terms of its asymptotic running time) is the *general number field sieve*. Heuristically, this algorithm runs in time $2^{\mathcal{O}(n^{1/3} \cdot (\log n)^{2/3})}$ on average to factor a number N of length $\mathcal{O}(n)$.[2]

8.1.1 Pollard's $p-1$ Method

In the case of integers $N = pq$ where $p - 1$ has only "small" factors, an algorithm due to Pollard can be used to efficiently factor (our description of the algorithm relies on some results proven in Appendix B.3.1). The key to

[1] If $f(n) = 2^{\Omega(n)}$, then f is exponential in n. If $f(n) = 2^{o(n)}$, then f is sub-exponential in n. A polynomial in n has the form $f(n) = 2^{\mathcal{O}(\log n)} = n^{\mathcal{O}(1)}$.

[2] It remains open to rigorously analyze the running time of this algorithm.

this approach is the following observation: Say we can find an element $y \in \mathbb{Z}_N^*$ for which $y \leftrightarrow (1, y_q)$ and $y_q \neq 1$. That is,

$$y = 1 \bmod p \quad \text{but} \quad y \neq 1 \bmod q \tag{8.1}$$

or, equivalently,

$$y - 1 = 0 \bmod p \quad \text{but} \quad y - 1 \neq 0 \bmod q.$$

The above means that $p \mid (y-1)$ but $q \nmid (y-1)$, which in turn implies that $\gcd(y-1, N) = p$. Thus, a simple gcd computation (which can be performed efficiently as described in Appendix B.1.2) yields a non-trivial factor of N.

The problem of factoring N has thus been reduced to finding a value y with the stated properties. We now describe how to find such a y. Say we have an integer B for which

$$(p-1) \mid B \quad \text{but} \quad (q-1) \nmid B. \tag{8.2}$$

(We defer until later the details of how such a B is determined.) Write $B = \gamma \cdot (p-1)$ for some integer γ. The algorithm (see the pseudocode below) chooses a random element $x \leftarrow \mathbb{Z}_N^*$ and sets $y := [x^B \bmod N]$; note that y can be computed using the efficient exponentiation algorithm from Appendix B.2.3.

ALGORITHM 8.1
Pollard's $p-1$ algorithm for factoring

Input: Integer N
Output: A non-trivial factor of N

$x \leftarrow \mathbb{Z}_N^*$
$y := [x^B \bmod N]$
$p := \gcd(y-1, N)$
if $p \notin \{1, N\}$ **return** p

We now show that y satisfies Equation (8.1) — and thus the algorithm finds a non-trivial factor of N — with relatively high probability. We have

$$\begin{aligned} y = [x^B \bmod N] \leftrightarrow (x_p, x_q)^B &= (x_p^B \bmod p,\ x_q^B \bmod q) \\ &= ((x_p^{p-1})^\gamma \bmod p,\ x_q^B \bmod q) = (1,\ x_q^B \bmod q) \end{aligned}$$

using Theorem 7.14 and the fact that the order of $\mathbb{Z}_p^*$ is $p-1$. That is, $y \leftrightarrow (1, x_q^B \bmod q)$. We thus see that y satisfies Equation (8.1) whenever $x_q^B \neq 1 \bmod q$.

Since q is prime, $\mathbb{Z}_q^*$ is a cyclic group of order $q-1$ that contains exactly $\phi(q-1)$ generators each of whose order is, by definition, $q-1$. (See Theorem B.18.) We claim that if x_q is a generator of $\mathbb{Z}_q^*$ and $(q-1) \nmid B$, then

$x_q^B \neq 1 \bmod q$. To see this, use division-with-remainder (Proposition 7.1) to write $B = \alpha \cdot (q-1) + \beta$ with $1 \leq \beta < (q-1)$. Then

$$x_q^B = x_q^{\alpha \cdot (q-1)+\beta} = (x_q^{q-1})^\alpha x_q^\beta = x_q^\beta \bmod q,$$

using Theorem 7.14. But $x_q^\beta \neq 1 \bmod q$ since the order of x_q is strictly larger than β. It remains to analyze the probability that x_q is a generator.

Assume $(q-1) \nmid B$. If x is chosen uniformly at random from $\mathbb{Z}_N^*$, then $x_q \stackrel{\text{def}}{=} [x \bmod q]$ is uniformly distributed in $\mathbb{Z}_q^*$. (This is a consequence of the fact that the Chinese remainder theorem gives a bijection between $\mathbb{Z}_N^*$ and $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.) Since $\mathbb{Z}_q^*$ has $\phi(q-1)$ generators, the probability of choosing x such that $x_q^B \neq 1 \bmod q$ is thus at least $\frac{\phi(q-1)}{q-1}$. Using Theorem B.16, this probability is $\Omega(1/\log q)$.

The preceding discussion shows that Pollard's $p-1$ algorithm succeeds in finding a non-trivial factor of N with probability $\Omega(1/\log q) = \Omega(1/n)$, assuming a B satisfying Equation (8.2) is known. (Alternatively, we can run the algorithm for $\mathcal{O}(\log^2 q) = \mathsf{poly}(n)$ iterations and find a non-trivial factor of N with all but negligible probability.) It remains to choose a value for B.

One possibility is to choose

$$B = \prod_{i=1}^{k} p_i^{\lfloor n/\log p_i \rfloor},$$

where p_i denotes the i^{th} prime (that is, $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$) and k is a bound whose choice affects both the running time and the success probability of the algorithm. Note that $p_i^{\lfloor n/\log p_i \rfloor}$ is the largest power of p_i that can divide $p-1$, an integer of length at most n. Thus, as long as $p-1$ can be written as $\prod_{i=1}^{k} p_i^{e_i}$ with $e_i \geq 0$ (that is, as long as $p-1$ has no prime factors larger than p_k), it will be the case that $(p-1) \mid B$. In contrast, if $q-1$ has any prime factor larger than p_k then $(q-1) \nmid B$.

Choosing a larger value for k increases B and so increases the running time of the algorithm (which performs a modular exponentiation to the power B). A larger value of k also makes it more likely that $(p-1) \mid B$ but at the same time makes it less likely that $(q-1) \nmid B$. It is, of course, possible to run the algorithm repeatedly using multiple choices for k. Other ways of selecting B have also been considered.

Pollard's $p-1$ method is thwarted if both $p-1$ and $q-1$ have only large prime factors. (More precisely, the algorithm still works but only when B is so large that the algorithm is no longer efficient.) For this reason, when generating a modulus $N = pq$ for cryptographic applications, p and q are sometimes chosen to be *strong* primes (recall that p is a strong prime if $(p-1)/2$ is also prime). Selecting p and q in this way is markedly less efficient than simply choosing p and q as *arbitrary* (random) primes. Because better factoring algorithms are available anyway, as we will see below, the current consensus

is that the added computational cost of generating p and q as strong primes is not offset by any appreciable security gains. However, we remark that certain cryptographic schemes (that we will not see in this book) require p and q to be strong primes for technical reasons related to the group structure of $\mathbb{Z}_N^*$.

8.1.2 Pollard's Rho Method

Unlike the algorithm of the previous section, Pollard's rho method can be used to find a non-trivial factor of an arbitrary integer N without assumptions regarding p or q; that is, it is a *general-purpose* factoring algorithm. Proving rigorous bounds on the running time/success probability of the algorithm is still an open question. However, heuristically, the algorithm factors N with constant probability in $\mathcal{O}\left(N^{1/4} \cdot \mathsf{polylog}(N)\right) \approx 2^{\|N\|/4}$ steps, an improvement over trial division.

The idea of the rho method is to find two distinct values $x, x' \in \mathbb{Z}_N^*$ that are equivalent modulo p (i.e., $x = x' \bmod p$); let us call such a pair of values *good*. Similarly to the previous section, we may observe that $x - x' = 0 \bmod p$ but $x - x' \neq 0 \bmod N$, and so $p \mid (x - x')$ but $N \nmid (x - x')$. But this means that $\gcd(x - x', N) = p$, a non-trivial factor of N.

How can we find a good pair? Assume we choose values $x_1, \ldots, x_k$ independently and uniformly at random from $\mathbb{Z}_N^*$, where $k = 2^{n/2} = \mathcal{O}(\sqrt{p})$. Using the birthday bounds proved in Appendix A.4, it follows that:

- The probability that there exist distinct i, j with $x_i = x_j$ is at most

$$\frac{k^2}{2 \cdot \phi(N)} = \frac{2^n}{2 \cdot \phi(N)} = \mathcal{O}(2^{-n}),$$

  which is negligible in n. (See Lemma A.9.)

- As a consequence of the bijectivity between $\mathbb{Z}_N^*$ and $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$ guaranteed by the Chinese remainder theorem, the values $\{[x_m \bmod p]\}_{m=1}^k$ are independently and uniformly distributed in $\mathbb{Z}_p^*$. Using Lemma A.10, the probability that there exist i, j with $[x_i \bmod p] = [x_j \bmod p]$ or, equivalently, $x_i = x_j \bmod p$, is roughly $1/4$.

Combining the above, we see that with probability roughly $1/4$ there will exist i, j with x_i, x_j a good pair; i.e.,

$$x_i = x_j \bmod p \quad \text{but} \quad x_i \neq x_j \bmod N.$$

This pair can then be used to find a non-trivial factor of N as discussed earlier.

We can generate $k = \mathcal{O}(\sqrt{p})$ random elements of $\mathbb{Z}_N^*$ in $\mathcal{O}(\sqrt{p}) = \mathcal{O}(N^{1/4})$ time. *Testing* all pairs of elements in order to find a good pair, however, would require $\binom{k}{2} = \mathcal{O}(k^2) = \mathcal{O}(p) = \mathcal{O}(N^{1/2})$ time! (Note that since p is unknown we cannot simply compute the sequence $\hat{x}_i \stackrel{\text{def}}{=} [x_i \bmod p]$ and then sort the $\hat{x}_i$

to find a good pair. Instead, for all i, j we must compute $\gcd(x_i - x_j, N)$ to see whether this gives a non-trivial factor of N.) Without further optimizations, this will be no better than trial division.

Pollard's idea was to choose $x_1, \ldots, x_k, \ldots, x_{2k}$ in a recursive manner, choosing $x_1 \in \mathbb{Z}_N^*$ at random and then computing $x_m := F(x_{m-1}) \bmod N$ for some appropriate function F. (The choice of F is discussed below.) Instead of testing each pair x_i, x_j (for all $i, j \leq k$) to find a good pair, it now suffices to test x_i and x_{2i} for all $i \leq k$, as justified by the following claim.

CLAIM 8.2 *Let $x_1, \ldots$ be a sequence of values with $x_m = F(x_{m-1}) \bmod N$. If $x_i = x_j \bmod p$ with $i < j$, then there exists an $i' < j$ such that $x_{i'} = x_{2i'} \bmod p$.*

PROOF If $x_i = x_j \bmod p$, then the sequence $[x_i \bmod p], [x_{i+1} \bmod p], \ldots$ repeats with period $j - i$. (That is, for all $i' \geq i$ and integers $\delta \geq 0$ it holds that $x_{i'} = x_{i'+\delta(j-i)} \bmod p$.) Take i' to be the smallest multiple of $j - i$ that is greater than or equal to i; that is, $i' \stackrel{\text{def}}{=} (j - i) \cdot \lceil i/(j - i) \rceil$. We must have $i' < j$ since the sequence $i, i + 1, \ldots i + (j - i - 1)$ contains a multiple of $j - i$. Since $2i' - i' = i'$ is a multiple of the period and $i' \geq i$, it follows that $x_{i'} = x_{2i'} \bmod p$. ■

By the claim above, if there is a good pair x_i, x_j in the sequence $x_1, \ldots, x_k$ then there is a good pair $x_{i'}, x_{2i'}$ in the sequence $x_1, \ldots, x_{2k}$. The number of pairs that need to be tested, however, is reduced from $\binom{k}{2}$ to $k = \mathcal{O}(\sqrt{p}) = \mathcal{O}(N^{1/4})$. A description of the entire algorithm follows.

ALGORITHM 8.3
Pollard's rho algorithm for factoring

Input: Integer N, a product of two n-bit primes
Output: A non-trivial factor of N

$x_0 \leftarrow \mathbb{Z}_N^*$
for $i = 1$ to $2^{n/2}$:
  $x_i := [F(x_{i-1}) \bmod N]$
  $x_{2i} := [F(F(x_{2i-2})) \bmod N]$
  $p := \gcd(x_{2i} - x_i, N)$
  if $p \notin \{1, N\}$ **return** p

Pollard's choice of $x_1, \ldots$ leads to an improvement in the running time of the algorithm. Unfortunately, since the values in the sequence are no longer chosen independently at random, the analysis given earlier (showing that a good pair exists with probability roughly $1/4$) no longer applies. Heuristically,

however, if the sequence "behaves randomly" then we expect that a good pair will still be found with probability roughly 1/4. (We stress that the sequence is certainly not pseudorandom in the sense of Chapter 3. However, cryptographic pseudorandomness is not a necessary condition for Pollard's rho algorithm to succeed.) Taking F of the form $F(x) = x^2 + b$, where $b \neq 0, -2 \bmod N$, gives an F that is efficient to compute and seems to work well in practice. (See [142, Section 10.2] for some rationale for this choice of F.) It remains an interesting open question to give a tight and rigorous analysis of Pollard's rho algorithm for any concrete F.

8.1.3 The Quadratic Sieve Algorithm

Pollard's rho algorithm runs in time exponential in the length of the number N to be factored. The *quadratic sieve* algorithm runs in sub-exponential time. It was the fastest-known factoring algorithm until the early '90s, and remains the factoring algorithm of choice for numbers up to about 300 bits long. We describe the general principles underlying the quadratic sieve algorithm but caution the reader that many important details are omitted.

Recall that an element $z \in \mathbb{Z}_N^*$ is a *quadratic residue modulo* N if there exists an $x \in \mathbb{Z}_p^*$ such that $x^2 = z \bmod N$; we say that x is a *square root of* z in this case. The following observations, used also in Chapter 11, serve as our starting point:

- If $N = pq$ is a product of two distinct primes, then every quadratic residue modulo N has exactly four square roots. See Section 11.1.2 for proof.
- Given x, y with $x^2 = y^2 \bmod N$ and $x \neq \pm y \bmod N$, it is possible to compute a non-trivial factor of N in polynomial time. This is by virtue of the fact that $x^2 = y^2 \bmod N$ implies

$$0 = x^2 - y^2 = (x - y)(x + y) \bmod N,$$

  and so $N \mid (x - y)(x + y)$. However, $N \nmid (x - y)$ and $N \nmid (x + y)$ because $x \neq \pm y \bmod N$. So it must be the case that $\gcd(x - y, N)$ is equal to one of the prime factors of N. See also Lemma 11.21.

The quadratic sieve algorithm tries to generate a pair of values x, y whose squares are equal modulo N; the hope is that with constant probability it will also hold that $x \neq \pm y \bmod N$. It searches for x and y via the following two-step process:[3]

Step 1. Fix a set $B = \{p_1, \ldots, p_k\}$ of small prime numbers. Find $\ell > k$ distinct values $x_1, \ldots, x_\ell \in \mathbb{Z}_N^*$ for which $q_i \stackrel{\text{def}}{=} [x_i^2 \bmod N]$ is "small", so that

[3]Some details have been changed in order to simplify the presentation. The description is merely meant to convey the main ideas of the algorithm.

q_i can be factored over the integers (using, e.g., trial division) and such that all the prime factors of q_i lie in B. (It is also required that $x_i > \sqrt{n}$, thus ensuring that $x_i^2 > n$ and the modular reduction of x_i^2 is not trivial.) We omit the details of how these $\{x_i\}$ are found.

Following this step, we have a set of equations of the form:

$$\begin{aligned} x_1^2 &= \prod_{i=1}^{k} p_i^{e_{1,i}} \bmod N \\ &\vdots \\ x_\ell^2 &= \prod_{i=1}^{k} p_i^{e_{\ell,i}} \bmod N. \end{aligned} \tag{8.3}$$

Reducing the exponents of each p_i modulo 2, we obtain the matrix Γ defined as

$$\begin{pmatrix} \gamma_{1,1} & \gamma_{1,2} & \cdots & \gamma_{1,k} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{\ell,1} & \gamma_{\ell,2} & \cdots & \gamma_{\ell,k} \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} [e_{1,1} \bmod 2] & [e_{1,2} \bmod 2] & \cdots & [e_{1,k} \bmod 2] \\ \vdots & \vdots & \ddots & \vdots \\ [e_{\ell,1} \bmod 2] & [e_{\ell,2} \bmod 2] & \cdots & [e_{\ell,k} \bmod 2] \end{pmatrix}.$$

The goal is to have $\ell > k$ with none of the rows of Γ all 0. Once this is accomplished, proceed to the next step.

Step 2. The matrix Γ constructed in the previous step has more rows than columns. Therefore, some subset of the rows must sum to the all-0 row modulo 2. (Furthermore, by construction, Γ has no all-0 rows.) An appropriate set of rows can be found efficiently using linear algebra. For the sake of illustration, say rows ℓ_1, ℓ_2, ℓ_3 sum to the all-0 row; that is,

$$\begin{array}{rccc} & \gamma_{\ell_1,1} & \cdots & \gamma_{\ell_1,k} \\ & \gamma_{\ell_2,1} & \cdots & \gamma_{\ell_2,k} \\ + & \gamma_{\ell_3,1} & \cdots & \gamma_{\ell_3,k} \\ \hline & 0 & \cdots & 0 \end{array}$$

where addition is modulo 2. Taking the appropriate equations from Equation (8.3), we have

$$X \stackrel{\text{def}}{=} x_{\ell_1}^2 \cdot x_{\ell_2}^2 \cdot x_{\ell_3}^2 = \prod_{i=1}^{k} p_i^{e_{\ell_1,i}+e_{\ell_2,i}+e_{\ell_3,i}} \bmod N.$$

Moreover, by choice of ℓ_1, ℓ_2, ℓ_3 we know that $e_{\ell_1,i} + e_{\ell_2,i} + e_{\ell_3,i}$ is even for all i. This means that we can write

$$X = (x_{\ell_1} \cdot x_{\ell_2} \cdot x_{\ell_3})^2 = \left(\prod_{i=1}^{k} p_i^{(e_{\ell_1,i}+e_{\ell_2,i}+e_{\ell_3,i})/2} \right)^2 \bmod N,$$

and we have found two elements whose squares are equal modulo N. Although there is no guarantee that $x_{\ell_1} \cdot x_{\ell_2} \cdot x_{\ell_3} \neq \pm \prod_{i=1}^{k} p_i^{(e_{\ell_1,i}+e_{\ell_2,i}+e_{\ell_3,i})/2} \bmod N$, we can at least heuristically expect that this will be the case with probability roughly $1/2$ (since X has four square roots).

Example 8.4
Take $N = 377753$. We have $6647 = [620^2 \bmod N]$, and we can factor 6647 (over the integers, without any modular reduction) as

$$6647 = 17^2 \cdot 23.$$

Thus, $620^2 = 17^2 \cdot 23 \bmod N$. Similarly,

$$\begin{aligned} 621^2 &= 2^4 \cdot 17 \cdot 29 \bmod N \\ 645^2 &= 2^7 \cdot 13 \cdot 23 \bmod N \\ 655^2 &= 2^3 \cdot 13 \cdot 17 \cdot 29 \bmod N. \end{aligned}$$

So

$$\begin{aligned} 620^2 \cdot 621^2 \cdot 645^2 \cdot 655^2 &= 2^{14} \cdot 13^2 \cdot 17^4 \cdot 23^2 \cdot 29^2 \bmod N \\ \Rightarrow [620 \cdot 621 \cdot 645 \cdot 655 \bmod N]^2 &= \left[2^7 \cdot 13 \cdot 17^2 \cdot 23 \cdot 29 \bmod N\right]^2 \bmod N \\ \Rightarrow 127194^2 &= 45335^2 \bmod N, \end{aligned}$$

with $127194 \neq \pm 45335 \bmod N$. Computing $\gcd(127194 - 45335, 377753) = 751$ yields a non-trivial factor of N. ♢

Running time. We have omitted many details in our discussion of the algorithm above. It can be shown, however, that with appropriate optimizations the quadratic sieve algorithm runs in time $2^{\mathcal{O}\left(\sqrt{n \cdot \log n}\right)}$ to factor a number N of length $\mathcal{O}(n)$. The important point is that this running time is sub-exponential in the length of N.

8.2 Algorithms for Computing Discrete Logarithms

Let $\mathbb{G}$ be a group for which the group operation can be carried out efficiently. By the results of Appendix B.2.3, this means that exponentiation in $\mathbb{G}$ can also be done efficiently. An *instance* of the discrete logarithm problem takes the following form (see Section 7.3.2): given $g \in \mathbb{G}$ and $y \in \langle g \rangle$, find x such that $g^x = y$. (Recall that $\langle g \rangle$, the cyclic subgroup generated by g, is the subgroup $\{g^0, g^1, \ldots\} \subseteq \mathbb{G}$. If $\langle g \rangle = \mathbb{G}$ then g is a *generator* of $\mathbb{G}$ and $\mathbb{G}$ is

cyclic.) This answer is denoted by $\log_g y$, and is uniquely defined modulo the order of g. We sometimes refer to g in an instance of the discrete logarithm problem as the *base*.

Algorithms for attacking the discrete logarithm problem fall into two categories: those that work for *arbitrary* groups (such algorithms are sometimes termed *generic*) and those that work for some *specific* group. For algorithms of the former type, we can often just as well take the group to be $\langle g \rangle$ itself (thus ignoring elements in $\mathbb{G} \setminus \langle g \rangle$ when g is not a generator of $\mathbb{G}$). When doing so, we will let q denote the order of $\langle g \rangle$ and assume that q is known. Note that brute-force search for the discrete logarithm $\log_g y$ can be done in time $\mathcal{O}(q)$, and so we will only be interested in algorithms whose running time is better than this.

We will discuss the following algorithms that work in arbitrary groups:

- The *baby-step/giant-step* method, due to Shanks, computes the discrete logarithm in a group of order q in time $\mathcal{O}(\sqrt{q} \cdot \mathsf{polylog}(q))$.
- The *Pohlig-Hellman* algorithm can be used when the factorization of the group order q is known. When q has small factors, this technique reduces the given discrete logarithm instance to multiple instances of the discrete logarithm problem in groups of smaller order. Solutions to each of the latter can be combined to give the desired solution to the original problem.

Following this, we will look at computing discrete logarithms in some specific groups. As an illustrative but simple example, in Section 8.2.3 we look at the problem in the (additive) group $\mathbb{Z}_N$ and show that discrete logarithms can be computed in polynomial time in this case.

> *The point of this example is to demonstrate that even though any cyclic group of order q is isomorphic to $\mathbb{Z}_q$ (cf. Example 7.58), and hence all cyclic groups of the same order are algebraically identical, the hardness of the discrete logarithm problem depends in a crucial way on the particular* **representation** *of the group being used.*

Indeed, the algorithm for computing discrete logarithms in the *additive* group $\mathbb{Z}_N$ will rely on the fact that *multiplication* modulo N is also defined. Such a statement makes no sense in some arbitrary group that is defined without reference to modular arithmetic.

Turning to groups with more cryptographic significance, in Section 8.2.4 we briefly discuss the computation of discrete logarithms in the cyclic group $\mathbb{Z}_p^*$ for p prime. We give a high-level overview of the *index calculus method* that solves the discrete logarithm problem in such groups in sub-exponential time. The full details of this approach are, unfortunately, beyond the scope of this book.

The baby-step/giant-step algorithm is known to be *optimal* in terms of its asymptotic running time as far as generic algorithms go. (We remark,

however, that more space-efficient generic algorithms with the same running time are known.) The proven lower bound on the complexity of finding discrete logarithms when the group is treated generically, however, says nothing about the hardness of finding discrete logarithms in any particular group, as illustrated by the ease of computing discrete logarithms in $\mathbb{Z}_N$.

Currently, the best-known algorithm for computing discrete logarithms in $\mathbb{Z}_p^*$ (for p prime) is the *general number field sieve*.[4] Heuristically, this algorithm runs in time $2^{\mathcal{O}(n^{1/3}\cdot(\log n)^{2/3})}$ on average to compute discrete logarithms in $\mathbb{Z}_p^*$ when p has length $\|p\| = \mathcal{O}(n)$. Importantly, essentially no non-generic algorithms are currently known for computing discrete logarithms in certain specially-constructed elliptic curve groups (cf. Section 7.3.4). This means that for such groups, as long as the group order is prime (so as to preclude using the Pohlig-Hellman algorithm), only exponential-time algorithms for computing discrete logarithms are known.

To get a sense for the practical importance of this latter remark, we can compare the group sizes needed for each type of group in order to make the discrete logarithm problem equally hard. (This will be a rough comparison only, as a more careful comparison would, for starters, need to take into account the constants implicit in the big-$\mathcal{O}$ notation of the running times given above.) For a 1024-bit prime p, the general number field sieve computes discrete logarithms in $\mathbb{Z}_p^*$ in roughly $e^{1024^{1/3}\cdot 10^{2/3}} \approx e^{10\cdot 4.6} = e^{46} \approx 2^{66}$ steps. This matches the time needed to compute discrete logarithms using the best generic algorithm in an elliptic curve group of order q, where q is a 132-bit prime, since then $\sqrt{q} \approx 2^{132/2} = 2^{66}$. We see that a significantly smaller elliptic curve group, with concomitantly faster group operations, can be used without reducing the difficulty of the discrete logarithm problem (at least with respect to the best currently-known techniques). Roughly speaking, then, by using elliptic curve groups in place of $\mathbb{Z}_p^*$ we obtain cryptographic schemes that are more efficient for the honest parties, but that are equally hard for an adversary to break. This explains why they have become popular, especially when implemented on weak hardware.

8.2.1 The Baby-Step/Giant-Step Algorithm

The baby-step/giant-step algorithm, due to Shanks, computes discrete logarithms in a group of order q in time $\mathcal{O}(\sqrt{q}\cdot \mathsf{polylog}(q))$. The idea is simple. Given input g and $y \in \langle g\rangle$, we can imagine the elements of $\langle g\rangle$ laid out in a circle as

$$1 = g^0, g^1, g^2, \ldots, g^{q-2}, g^{q-1}, g^q = 1,$$

and we know that y must lie somewhere on this circle. Computing and writing down all the points on this circle would take $\Omega(q)$ time. Instead, we "mark

[4] It is no accident that this name is shared by algorithms for factoring and for computing discrete logarithms, since these algorithms share many of the same underlying steps.

off" the circle at intervals of size $t \stackrel{\text{def}}{=} \lfloor \sqrt{q} \rfloor$; that is, we compute and record the $\lfloor q/t \rfloor + 1 = \mathcal{O}(\sqrt{q})$ elements

$$g^0, g^t, g^{2t}, \ldots, g^{\lfloor q/t \rfloor \cdot t}.$$

(These are the "giant steps".) Note that the "gap" between any consecutive "marks" on the circle is at most t. Furthermore, we know that $y = g^x$ lies in one of these gaps. We are thus guaranteed that one of the t elements

$$y \cdot g^0 = g^x, \;\; y \cdot g^1 = g^{x+1}, \;\; \ldots, \;\; y \cdot g^t = g^{x+t},$$

will be equal to one of the points we have marked off. (These are the "baby steps".) Say $y \cdot g^i = g^{k \cdot t}$. We can easily solve this to obtain $y = g^{kt-i}$ or $\log_g y = [kt - i \bmod q]$. Pseudocode for this algorithm follows.

ALGORITHM 8.5
The baby-step/giant-step algorithm

Input: Elements $g \in \mathbb{G}$ and $y \in \langle g \rangle$; the order q of g
Output: $\log_g y$

$t := \lfloor \sqrt{q} \rfloor$
for $i = 0$ to $\lfloor q/t \rfloor$:
  compute $g_i := g^{i \cdot t}$
sort the pairs (i, g_i) by their second component
for $i = 0$ to t:
  compute $y_i := y \cdot g^i$
  if $y_i = g_k$ for some k, **return** $[kt - i \bmod q]$

The algorithm requires $\mathcal{O}(\sqrt{q})$ exponentiations and multiplications in $\mathbb{G}$, and each exponentiation can be done in time $\mathcal{O}(\mathsf{polylog}(q))$ using an efficient exponentiation algorithm. (Actually, other than the first value $g_1 = g^t$, each value g_i can be computed using a single multiplication as $g_i := g_{i-1} \cdot g_1$. Similarly, each y_i can be computed as $y_i := y_{i-1} \cdot g$.) Sorting the $\mathcal{O}(\sqrt{q})$ pairs (i, g_i) can be done in time $\mathcal{O}(\sqrt{q} \cdot \log q)$, and we can then use binary search to check whether y_i is equal to some g_k in time $\mathcal{O}(\log q)$. The overall algorithm thus runs in time $\mathcal{O}(\sqrt{q} \cdot \mathsf{polylog}(q))$.

Example 8.6
We show an application of the algorithm in the cyclic group $\mathbb{Z}_{29}^*$ of order $q = 29 - 1 = 28$. Take $g = 2$ and $y = 17$. We set $t = 5$ and compute:

$$2^0 = 1, \;\; 2^5 = 3, \;\; 2^{10} = 9, \;\; 2^{15} = 27, \;\; 2^{20} = 23, \;\; 2^{25} = 11.$$

(It should be understood that all operations are in $\mathbb{Z}_{29}^*$.) Then compute:

$$17 \cdot 2^0 = 17, \quad 17 \cdot 2^1 = 5, \quad 17 \cdot 2^2 = 10, \quad 17 \cdot 2^3 = 20, \quad 17 \cdot 2^4 = 11, \quad 17 \cdot 2^5 = 22,$$

and notice that $2^{25} = 11 = 17 \cdot 2^4$. We thus have $\log_2 17 = 25 - 4 = 21$. $\diamondsuit$

8.2.2 The Pohlig-Hellman Algorithm

The Pohlig-Hellman algorithm can be used to speed up the computation of discrete logarithms when any non-trivial factors of the group order q are known. Recall that the order of an element g, which we denote here by $\mathsf{ord}(g)$, is the smallest positive i for which $g^i = 1$. We will need the following lemma:

LEMMA 8.7 *Let* $\mathsf{ord}(g) = q$, *and say* $p \mid q$. *Then* $\mathsf{ord}(g^p) = q/p$.

PROOF Since $(g^p)^{q/p} = g^q = 1$, the order of g^p is certainly at most q/p. Let $i > 0$ be such that $(g^p)^i = 1$. Then $g^{pi} = 1$ and, since q is the order of g, it must be the case that $pi \geq q$ or equivalently $i \geq q/p$. The order of g^p is therefore exactly q/p. ■

We will also use a generalization of the Chinese remainder theorem: if $q = \prod_{i=1}^{k} q_i$ and the $\{q_i\}$ are pairwise relatively prime (i.e., $\gcd(q_i, q_j) = 1$ for all $i \neq j$), then

$$\mathbb{Z}_q \simeq \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_k} \quad \text{and} \quad \mathbb{Z}_q^* \simeq \mathbb{Z}_{q_1}^* \times \cdots \times \mathbb{Z}_{q_k}^*.$$

(This can be proved by induction on k, using the basic Chinese remainder theorem as the base case.) Moreover, by an extension of the algorithm in Section 7.1.5 it is possible to convert efficiently between the representation of an element as an element of $\mathbb{Z}_q$ and its representation as an element of $\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_k}$.

We now describe the Pohlig-Hellman approach. We are given g and y and wish to find an x such that $g^x = y$. Let $\mathsf{ord}(g) = q$, and say a factorization

$$q = \prod_{i=1}^{k} q_i$$

is known with the $\{q_i\}$ pairwise relatively prime. (Note that this need not be the complete prime factorization of q.) We know that

$$\left(g^{q/q_i}\right)^x = (g^x)^{q/q_i} = y^{q/q_i} \quad \text{for } i = 1, \ldots, k. \tag{8.4}$$

Letting $g_i \stackrel{\text{def}}{=} g^{q/q_i}$, we thus have k instances of a discrete logarithm problem in k *smaller* groups, each of size $\mathsf{ord}(g_i) = q_i$ (by Lemma 8.7).

We can solve each of the k resulting instances using any other algorithm for solving the discrete logarithm problem; for concreteness, let us assume that the baby-step/giant-step algorithm of the previous section is used. Solving these instances gives a set of answers $\{x_i\}_{i=1}^{k}$ for which $g_i^{x_i} = y^{q/q_i} = g_i^x$. (The second equality follows from Equation (8.4).) Proposition 7.50 implies

that $x = x_i \bmod q_i$ for all i. By the generalized Chinese remainder theorem discussed earlier, the constraints

$$\begin{aligned} x &= x_1 \bmod q_1 \\ &\vdots \\ x &= x_k \bmod q_k \end{aligned}$$

uniquely determine x modulo q. Thus, x can be efficiently reconstructed from $x_1, \ldots, x_k$, as required.

Example 8.8
We apply the ideas introduced here to again compute a discrete logarithm in $\mathbb{Z}_p^*$. Here, take $p = 31$ with the order of $\mathbb{Z}_{31}^*$ being $q = 31 - 1 = 30 = 5 \cdot 3 \cdot 2$. Say $g = 3$ and $y = 26 = g^x$. We have:

$$\begin{aligned} (g^{30/5})^x = y^{30/5} &\Rightarrow (3^6)^x = 26^6 \Rightarrow 16^x = 1 \\ (g^{30/3})^x = y^{30/3} &\Rightarrow (3^{10})^x = 26^{10} \Rightarrow 25^x = 5 \\ (g^{30/2})^x = y^{30/2} &\Rightarrow (3^{15})^x = 26^{15} \Rightarrow 30^x = 30. \end{aligned}$$

(Once again, we omit the "mod 31" since this is understood.) Solving each equation, we obtain

$$x = 0 \bmod 5, \quad x = 2 \bmod 3, \quad x = 1 \bmod 2,$$

and so $x = 5 \bmod 30$. Indeed, $3^5 = 26 \bmod 31$. ♢

Assuming q with factorization as above, and assuming the baby-step/giant-step algorithm is used to solve each of the smaller instances of the discrete logarithm problem, the running time of the entire algorithm will be $\mathcal{O}(\mathsf{polylog}(q) \cdot \sum_{i=1}^{k} \sqrt{q_i})$. Since q can have at most $\log q$ factors, this simplifies to $\mathcal{O}(\mathsf{polylog}(q) \cdot \max_i\{\sqrt{q_i}\})$. Depending on the size of the largest known factor of q, this can be a marked improvement over the $\mathcal{O}(\sqrt{q})$ algorithm given in the previous section. In particular, if q has many small factors then the discrete logarithm problem in a group of order q will be relatively easy to solve via this approach. As discussed in Section 7.3.2, this motivates choosing q to be prime for cryptographic applications.

If q has prime factorization $q = \prod_{i=1}^{k} p_i^{e_i}$, the Pohlig-Hellman algorithm as described above solves the discrete logarithm problem in a group of order q in time $\mathcal{O}\left(\mathsf{polylog}(q) \cdot \max_i\{\sqrt{p_i^{e_i}}\}\right)$. Using additional ideas, this can be improved to $\mathcal{O}\left(\mathsf{polylog}(q) \cdot \max_i\{\sqrt{p_i}\}\right)$; see Exercise 8.3.

8.2.3 The Discrete Logarithm Problem in $\mathbb{Z}_N$

The algorithms shown in the preceding two sections are *generic*, in the sense that they are oblivious to the underlying group in which the discrete logarithm

problem is defined (except for knowledge of the group order). The purpose of this brief section is merely to emphasize that non-generic algorithms, which make use of the *particular* (representation of the) group under consideration, can potentially perform much better.

Consider the task of computing discrete logarithms in the (additive) group $\mathbb{Z}_N$ for arbitrary N. The problem is trivial with respect to the base $g = 1$: the discrete logarithm of element $y \in \mathbb{Z}_N$ is simply the integer y itself since $y \cdot 1 = y \bmod N$. Note that, formally speaking, the 'y' on the left-hand side of this equation denotes the integer y while the 'y' on the right-hand side denotes the element $y \in \mathbb{Z}_N$. Nevertheless, the particular nature of the group $\mathbb{Z}_N$ allows us to essentially view these two instances of 'y' interchangeably.

Things are only mildly more complicated if a generator $g \neq 1$ is used. (Exercise 8.4 deals with the case when g is not a generator of $\mathbb{Z}_N$.) Let $g \in \mathbb{Z}_N$ be a generator and say we want to compute x such that $x \cdot g = y \bmod N$ for some given value y. Using Theorem B.18 (along with the fact that 1 is a generator), we have $\gcd(g, N) = 1$. But then g has a multiplicative inverse g^{-1} modulo N (and this inverse can be computed efficiently as discussed in Appendix B.2.2). The desired solution is simply $x = y \cdot g^{-1} \bmod N$.

It is interesting to pinpoint once again exactly what non-generic properties of $\mathbb{Z}_N$ are being used here. In this case, the algorithm implicitly uses the fact that an operation (namely, multiplication modulo N) *other than* the group operation (i.e., addition modulo N) is defined on the elements of the group.

8.2.4 The Index Calculus Method

The index calculus method solves the discrete logarithm problem in the cyclic group $\mathbb{Z}_p^*$ (for p prime) in time that is sub-exponential in the length of p. The astute reader may notice that the algorithm as we will describe it bears some resemblance to the quadratic sieve factoring algorithm introduced in Section 8.1.3. As in the case of that algorithm, we discuss the main ideas used by the index calculus method but the details are beyond the scope of our treatment. Also, some simplifications are introduced to clarify the presentation.

The index calculus method uses a two-step process. Importantly, the first step requires knowledge only of the modulus p and the base g and so it can be run as a 'pre-processing step' before y — the value whose discrete logarithm we are interested in — is known. For the same reason, it suffices to run the first step only once in order to solve multiple instances of the discrete logarithm problem (as long as all these instances share the same p and g).

Step 1. Let $q = p - 1$, the order of $\mathbb{Z}_p^*$. Fix a set $B = \{p_1, \ldots, p_k\}$ of small prime numbers. In this step, we find $\ell \geq k$ distinct, non-zero values $x_1, \ldots, x_\ell \in \mathbb{Z}_q$ for which $g_i \stackrel{\text{def}}{=} [g^{x_i} \bmod p]$ is "small", so that g_i can be factored over the integers (using, e.g., trial division) and such that all the prime factors of g_i lie in B. We do not discuss how these $\{x_i\}$ are found.

Following this step, we have ℓ equations of the form:

$$\begin{aligned} g^{x_1} &= \prod_{i=1}^{k} p_i^{e_{1,i}} \bmod p \\ &\vdots \\ g^{x_\ell} &= \prod_{i=1}^{k} p_i^{e_{\ell,i}} \bmod p. \end{aligned}$$

Taking discrete logarithms of each side, we can transform these into linear equations:

$$\begin{aligned} x_1 &= \sum_{i=1}^{k} e_{1,i} \cdot \log_g p_i \mod (p-1) \\ &\vdots \\ x_\ell &= \sum_{i=1}^{k} e_{\ell,i} \cdot \log_g p_i \mod (p-1). \end{aligned} \tag{8.5}$$

Note that the $\{x_i\}$ and the $\{e_{j,i}\}$ are known, while the $\{\log_g p_i\}$ are unknown.

Step 2. Now we are given an element y and want to compute $\log_g y$. Here, we find a value $x^* \in \mathbb{Z}_q$ for which $g^* \stackrel{\text{def}}{=} [g^{x^*} \cdot y \bmod p]$ is "small", so that g^* can be factored over the integers and such that all the prime factors of g^* lie in B. We do not discuss how x^* is found.

Say

$$\begin{aligned} g^{x^*} \cdot y &= \prod_{i=1}^{k} p_i^{e_i^*} \bmod p \\ \Rightarrow x^* + \log_g y &= \sum_{i=1}^{k} e_i^* \cdot \log_g p_i \mod (p-1), \end{aligned}$$

where x^* and the $\{e_i^*\}$ are known. Combined with Equation (8.5), we have $\ell + 1 \geq k + 1$ linear equations in the $k + 1$ unknowns $\{\log_g p_i\}_{i=1}^{k}$ and $\log_g y$. Using linear-algebraic[5] methods (and assuming the system of equations is not under-defined), we can solve for each of the unknowns and in particular solve for the desired solution $\log_g y$.

[5] Technically, things are slightly more complicated since the linear equations are all modulo $p - 1$, which is not prime. Nevertheless, there exist techniques for dealing with this.

Example 8.9
Let $p = 101$, $g = 3$, and $y = 87$. We have $3^{10} = 65 \bmod 101$, and $65 = 5 \cdot 13$ (over the integers). Similarly, $3^{12} = 80 = 2^4 \cdot 5 \bmod 101$ and $3^{14} = 13 \bmod 101$. We thus have the linear equations

$$\begin{aligned} 10 &= \log_3 5 + \log_3 13 \bmod 100 \\ 12 &= 4 \cdot \log_3 2 + \log_3 5 \bmod 100 \\ 14 &= \log_3 13 \bmod 100. \end{aligned}$$

We also have $3^5 \cdot 87 = 32 = 2^5 \bmod 101$, or

$$5 + \log_3 87 = 5 \cdot \log_3 2 \bmod 100. \tag{8.6}$$

Adding the second and third equations and subtracting the first, we derive $4 \cdot \log_3 2 = 16 \bmod 100$. This doesn't determine $\log_3 2$ uniquely, but it does tell us that $\log_3 2 = 4, 29, 54$, or 79 (cf. Exercise 8.4). Trying all possibilities shows that $\log_3 2 = 29$. Plugging this into Equation (8.6) gives $\log_3 87 = 40$. ♢

Running time. It can be shown that with appropriate optimizations the index calculus algorithm runs in time $2^{\mathcal{O}(\sqrt{n \cdot \log n})}$ to compute discrete logarithms in $\mathbb{Z}_p^*$ for p a prime of length n. The important point is that this is sub-exponential in $\|p\|$. Note that the expression for the running time is identical to that of the quadratic sieve method.

References and Additional Reading

The $p - 1$ method for factoring was described by Pollard in 1974 [115], and his rho method was described the following year [116]. (He later published a related rho method for computing discrete logarithms.) The quadratic sieve algorithm was introduced by Pomerance [117], based on earlier ideas of Dixon [49].

The baby-step/giant-step approach to computing discrete logarithms is due to Shanks [126], and the Pohlig-Hellman algorithm was published in 1978 [114]. For a discussion of the index calculus method, as well as an overview of methods for computing discrete logarithms, the reader is referred to the survey by Odlyzko [112].

The texts by Wagstaff [142], Shoup [131], Crandall and Pomerance [40], and Bressoud [29] all provide further information regarding algorithms for factoring and computing discrete logarithms.

Lower bounds on so-called *generic algorithms* for computing discrete logarithms (i.e., algorithms that apply to arbitrary groups without regard for the way the group is represented) are given by Nechaev [111] and Shoup [128].

Lenstra and Verheul [94] provide a comprehensive discussion of how the known algorithms for factoring and computing discrete logarithms affect the choice of cryptographic parameters in practice.

Exercises

8.1 In order to speed up the key generation algorithm for RSA, it has been suggested to generate a prime by generating many small random primes, multiplying them together and adding one (of course, then checking that the result is prime). Ignoring the question of the probability that such a value really is prime, what do you think of this method?

8.2 Here we show how to solve the discrete logarithm problem in a cyclic group of order $q = p^e$ in time $\mathcal{O}(\mathsf{polylog}(q) \cdot \sqrt{p}\,)$. Given as input a generator g of known order p^e and a value y, we want to compute $x = \log_g y$. Note that p can be computed easily from q (see Exercise 7.11).

(a) Show how to find $[x \bmod p]$ in time $\mathcal{O}(\mathsf{polylog}(q) \cdot \sqrt{p}\,)$.
Hint: Solve the equation
$$\left(g^{p^{e-1}}\right)^{x_0} = y^{p^{e-1}}$$
and use the same ideas as in the Pohlig-Hellman algorithm.

(b) Say $x = x_0 + x_1 \cdot p + \cdots + x_{e-1} \cdot p^{e-1}$ with $0 \le x_i < p$. In the previous step we determined x_0. Show how to compute in $\mathsf{polylog}(q)$ time a value y_1 such that $(g^p)^{x_1 + x_2 \cdot p + \cdots + x_{e-1} \cdot p^{e-2}} = y_1$.

(c) Use recursion to obtain the claimed running time for the original problem. (Note that $e = \mathsf{polylog}(q)$.)

8.3 Let q have prime factorization $q = \prod_{i=1}^{k} p_i^{e_i}$. Using the result from the previous problem, show a modification of the Pohlig-Hellman algorithm that solves the discrete logarithm problem in a group of order q in time $\mathcal{O}\left(\mathsf{polylog}(q) \cdot \sum_{i=1}^{k} e_i \sqrt{p_i}\right) = \mathcal{O}\left(\mathsf{polylog}(q) \cdot \max\{\sqrt{p_i}\}\right)$.

8.4 (a) Show that if $ab = c \bmod N$ and $\gcd(b, N) = d$, then:
i. $d \mid c$;
ii. $a \cdot (b/d) = (c/d) \bmod (N/d)$; and
iii. $\gcd(b/d, N/d) = 1$.

(b) Describe how to use the above to compute discrete logarithms in $\mathbb{Z}_N$ efficiently even when the base g is not a generator of $\mathbb{Z}_N$.

8.5 Using a variant of Claim 8.2, fully describe and analyze the constant-space birthday attack on cryptographic hash functions that was briefly described in Section 4.6.3.

Chapter 9

Private-Key Management and the Public-Key Revolution

9.1 Limitations of Private-Key Cryptography

We have seen that private-key cryptography can be used to enable secure communication over an insecure channel. While it therefore appears to solve completely the primary problem of cryptography, we discuss here a number of reasons why that is not the case.

The Key-Distribution Problem

Private-key cryptography requires shared, secret keys between the communicating parties. We have not yet dealt at all with the question of how these shared keys are obtained in the first place. Clearly, these keys cannot simply be sent over an insecure communication channel, because an eavesdropping adversary could then observe them *en route*.

The initial sharing of a secret key can be done using a secure channel that can be implemented, e.g., using a trusted messenger service. This option is likely to be unavailable to the average person, though governments, the military, intelligence organizations, and other such entities do have the means to share keys in this way. (Indeed, it is rumored that the red phone connecting Moscow and Washington was encrypted using a one-time pad, where the keys were shared by diplomats who flew from one country to the other carrying briefcases full of print-outs of the pad.) A more pragmatic method for two parties to share a key is for these parties to arrange a physical meeting at which time a random key can be generated, and a copy of the key given to each party. Although one can imagine two users arranging such a meeting on a dark street corner, a more commonplace setting where this might take place is a standard work environment. For example, a manager might share a key with each employee on the day when that employee first shows up at work.

While this might be a viable approach when only the manager shares keys with each employee, it does not scale well when all employees are required to share keys with each other. Extending the above approach would mean that every time a new employee arrived, *all other employees* would have to share a

new secret key with her. This would be especially problematic if the company were large and had offices in a number of different physical locations.

A partial solution in this setting is to use a designated "controller" (say, the IT manager of the company) to establish shared keys between all employees. Specifically, when a new employee joins the company the controller could generate random keys $k_1, \ldots$, give these keys (in person) to the new employee, and then send key k_i to the ith existing employee by encrypting k_i using the secret key shared between the controller and this employee. (We assume here that the controller is an employee, and so all existing employees share a key with him.) This is a very cumbersome approach. More importantly, it does not give a complete solution since keys are not completely secret. A dishonest controller could decrypt all inter-employee communication, and if an adversary ever compromised the controller's computer then all keys in the system would be revealed.

Key Storage and Secrecy

Consider again the aforementioned work environment where each pair of employees shares a secret key. When there are U employees, the number of secret keys in the system is $\binom{U}{2} = \Theta(U^2)$. More importantly, this means that every employee holds $U-1$ secret keys. In fact, the situation may be far worse because employees may also need keys in order to communicate securely with *remote resources* such as databases, servers, and so on. When the organization in question is large this creates a huge problem, on a number of levels. First, the proliferation of many secret keys is a significant logistical problem. Second, all these secret keys must be stored securely. The more keys there are, the harder it is to protect them, and the higher the chance of some keys being stolen by an adversary. Computer systems are often infected by viruses, worms, and other forms of malicious software. These malicious programs can be instructed to steal secret keys and send them quietly over the network to an attacker; such programs have been deployed in the past and their existence is not only a theoretical threat. Thus, storing keys on a personal computer is not always a reasonable solution.

Potential compromise of secret keys is always a concern, irrespective of the number of keys each party holds. When only a few keys need to be stored, however, there are good solutions available for dealing with this threat. A typical solution today is to store keys on a *smartcard*, a highly-protected hardware device. The smartcard can carry out cryptographic computations using the stored secret keys, ensuring that these keys never make their way onto users' personal computers. Since smartcards are much more resilient to attack than personal computers — for example, they typically cannot be infected by a virus — this offers a good means of protecting users' secret keys. Unfortunately, smartcards are typically quite limited in memory, and so cannot store hundreds (or thousands) of keys.

In principle, of course, it is possible to securely store any number of keys and the problem of secure storage can be solved by organizations, like governments, that can devote significant resources to the problem, though often at great inconvenience. The second author once spoke to someone who worked for the US embassy in a Western European country many years ago. His job was to decrypt all incoming communications, and the system was basically as follows: Whenever an encrypted message arrived, he took the message to a locked and guarded room where all secret keys were stored. He then found the appropriate key, and used that key to decrypt the message. The point of the story is that governments and large-scale organizations can potentially use private-key cryptography alone for securing their communication. However, such solutions are very costly, do not scale well, and are not suitable for settings that are typical for industry or for personal use.

Open Systems

As discussed above, private-key cryptography can be difficult to deploy and maintain, and requires the management and secure storage of a significant number of keys. At least in theory, however, it can be used to solve the problem of secure communication in "closed" systems where it is possible to distribute secret keys via physical means. Unfortunately, in "open" settings where parties have no way of securely distributing keys, private-key cryptography by itself is simply insufficient. For example, when encryption is needed for making a purchase over the Internet, or for sending email to a colleague in another country (whom the sender may never have met), private-key cryptography alone simply does not provide a solution. Due to its importance, we reiterate this point:

> *Solutions that are based on private-key cryptography are not sufficient to deal with the problem of secure communication in open systems where parties cannot physically meet, or where parties have transient interactions.*

This situation means that we must look further for adequate solutions.

9.2 A Partial Solution – Key Distribution Centers

As we have described, there are three distinct problems that arise with respect to the use of private-key cryptography. The first is that of key distribution; the second is that of managing so many secret keys; and the third is the inapplicability of private-key cryptography in open systems. Although it is impossible to fully solve the first problem, there is a solution that alleviates the first two problems and makes it feasible to implement private-key

solutions in large organizations. An extension of the idea (that we do not discuss further) allows private-key cryptography to be used in "partially-open" systems consisting of multiple organizations that mutually trust each other to some limited extent.

Key distribution centers. Consider again the case of a large organization where all pairs of employees must be able to communicate securely. The solution in which each pair of employees shares a key results in a huge proliferation of keys. A different approach is to rely on the fact that all employees may *trust* some entity — say, the IT manager of the organization — at least with respect to the security of work-related information. It is therefore possible for the IT manager to set up a single server, called a *key distribution center* (KDC), that can act as an *intermediary* between employees that wish to communicate. A KDC can work in the following way. First, all employees share a single key with the KDC; this key can be generated and shared, e.g., on the employee's first day at work. Then, when employee Alice wants to communicate securely with employee Bob, she sends a message to the KDC saying `'Alice wishes to communicate with Bob'` (where this message is authenticated using the key shared by Alice and the KDC). The KDC then chooses a new random secret key, called a *session key*, and sends this key to Alice encrypted using Alice's key, and also to Bob encrypted using Bob's key. Once Alice and Bob recover the session key, they can use it to communicate securely. When they are done with their conversation, they can (and should) erase this key because they can always contact the server again should they wish to communicate again at some later time. We remark that this is just a sketch of the solution and is not sufficient to provide the necessary level of security. (It is beyond the scope of this book to provide rigorous definitions and proofs for analyzing these solutions.) Nevertheless, it is enough to give a feeling of how to make private-key cryptography workable.

Consider the advantages of this approach:

1. Each employee needs to store only *one* secret key and so a smartcard-type solution can be deployed. It is true that the KDC needs to store many keys. However, the KDC can be secured in a safe place and given the highest possible protection against network attacks.

2. When an employee joins the organization all that must be done is to set up a secret key between this employee and the KDC. No other employees need to update the set of keys they hold. The same is true when an employee leaves the organization.

Thus, this approach alleviates two problems related to private-key cryptography: key distribution is simplified (only one new key must be shared when a party joins the organization), and key storage issues are resolved (only a single key needs to be stored by each party except the KDC). This approach makes private-key cryptography practical in a single organization, where there is one entity who is trusted by everyone.

Having said this, there are also some disadvantages to this approach:

1. A successful attack on the KDC will result in a complete break of the system for all parties. Thus, the motivation to break into the KDC is very great, increasing the security risk. In addition, an adversary internal to the organization who has access to the KDC (for example, the IT manager) can decrypt all communication between all parties.

2. The KDC is a single point of failure: if the KDC crashes, secure communication is temporarily impossible. Since all employees are continually contacting the KDC, the load on the KDC can be very high thereby increasing the chances that it may fall or be slow to respond.

   A simple solution is to replicate the KDC. This works (and is done in practice), but the existence of more KDCs means that there are now more points of attack on the system. Furthermore, it becomes more difficult to add or remove employees, since updates must be securely propagated to all KDCs.

The KDC-based solution above is similar to the solution we gave earlier whereby a designated "controller" sets up shared keys between all employees any time a new employee joins the organization. In the previous case, the controller is essentially acting as an *off-line* KDC. Since the controller is only involved in the initial setup, all employees still need to hold many secret keys. This is in contrast to the solution given here where the KDC is *online* and so can be used to interactively exchange keys between any pair of parties when needed. This means that each party needs to store only a single secret key.

Protocols for key distribution using a KDC. There are a number of protocols that can be found in the literature for secure key distribution using a KDC. One of these is the classic Needham-Schroeder protocol. We will not go into the details of this protocol (or any other) and instead refer the reader to the references listed at the end of this chapter for more details. We do mention one engineering feature of the protocol. When Alice contacts the KDC and asks to communicate with Bob, the KDC does not send the encrypted session key to both Alice and Bob. Rather, the KDC sends the session key encrypted under both Alice's *and Bob's* keys to Alice, and Alice herself forwards to Bob the session key encrypted under his key; see Figure 9.1. The protocol was designed in this way due to the fact that Bob may not be online; this could potentially cause a problem for the KDC who might "hang" indefinitely waiting for Bob to respond. By sending both encrypted keys to Alice, the KDC is relieved of maintaining an open session. The session key encrypted under Bob's key that the KDC sends to Alice is called a *ticket*, and can be viewed as a credential allowing Alice to talk to Bob.

We remark that using a CPA-secure encryption scheme to encrypt the session keys leaves the protocol vulnerable to attack; it is actually necessary to use a secure message transmission scheme as introduced in Section 4.9.

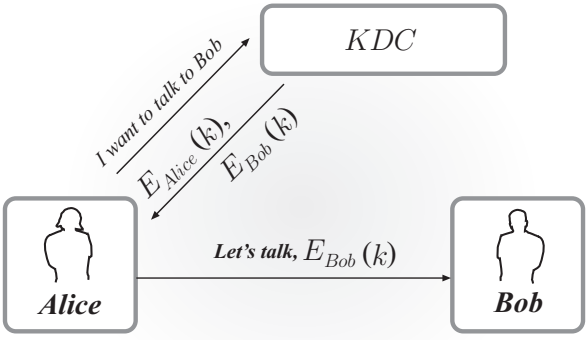

FIGURE 9.1: A general template for key-distribution protocols.

In practice, protocols like the Needham-Schroeder protocol are widely used. In many cases Alice and Bob might not both be users, but instead Alice might be a user and Bob a resource. For example, Alice may wish to read from a protected disk on some server. Alice asks for "permission" to do this from the KDC, who issues Alice a ticket that serves as Alice's credentials for reading from that disk. This ticket contains a session key (as described above) and thus Alice's communication with the server can be protected. A very widely-used system for implementing user authentication and secure communication via a KDC is the Kerberos protocol that was developed at MIT. Kerberos has a number of important features, and is the method used by Microsoft Windows (in Windows 2000 and above) for securing an internal network.

We conclude by noting that in practice the secret key that Alice shares with the KDC is often a short, easy-to-memorize password (because a typical user may not have a smartcard for storing long secret keys). In this case, many additional security problems arise that must be considered and dealt with. Once again, we refer the interested reader to the references listed at the end of this chapter for more information about such issues and how they are addressed.

9.3 The Public-Key Revolution

Key distribution centers and protocols like Kerberos are very useful, and are commonly used in practice. However, they still cannot solve the problem of key distribution in open systems like the Internet, where there are no private channels. (In the KDC setting, we implicitly assumed the existence of a private channel that was used to set up an initial key between the employees and the KDC.) To achieve private communication without ever communicating over a private channel, something radically different must be

used. In 1976, Whitfield Diffie and Martin Hellman published a paper with an innocent-looking title called "New Directions in Cryptography" [47]. The influence of this paper was enormous. In addition to introducing a fundamentally different way of looking at cryptography, it served as one of the first steps toward moving cryptography out of the private domain and into the public one. Before describing the basic ideas of Diffie and Hellman, we quote the first two paragraphs of their paper:

> *We stand today on the brink of a revolution in cryptography. The development of cheap digital hardware has freed it from the design limitations of mechanical computing and brought the cost of high grade cryptographic devices down to where they can be used in such commercial applications as remote cash dispensers and computer terminals.*
>
> *In turn, such applications create a need for new types of cryptographic systems which minimize the necessity of secure key distribution channels and supply the equivalent of a written signature. At the same time, theoretical developments in information theory and computer science show promise of providing provably secure cryptosystems, changing this ancient art into a science.*

Diffie and Hellman were not exaggerating, and the revolution they spoke of was due in great part to their work. Until 1976, it was well accepted that encryption simply could not be done without first sharing a secret key. However, Diffie and Hellman observed that there is a natural *asymmetry* in the world: that is, there are certain actions that can be easily performed but not easily reversed. For example, padlocks can be locked without a key (i.e., easily), but then cannot be reopened. More strikingly, it is easy to shatter a glass vase but extremely difficult to put it back together again. Algorithmically, and more to the point, it is easy to multiply two large primes but difficult to recover these primes from their product (this is exactly the factoring problem discussed in the previous chapter). The existence of such phenomena implies the possibility of constructing an encryption scheme that does not rely on shared secrets, but rather one for which encrypting is "easy" but *reversing* this operation (i.e., decrypting) is infeasible for anyone other than the designated receiver.

In a bit more detail, we can imagine a cryptosystem where there are *two* keys instead of one: one of these keys is an *encryption* key, used by senders to encrypt their messages, and the other is a *decryption* key, used by the receiver to recover the message from a ciphertext. Furthermore — and here it is amazing that something of this sort could possibly exist! — the secrecy of encrypted messages should be preserved *even against an adversary who knows the encryption key* (but not the decryption key). Encryption schemes with this property are called *asymmetric* or *public-key* encryption schemes, in contrast to the symmetric, or private-key, encryption schemes that we have seen so far. In a public-key encryption scheme the encryption key is called

the *public key*, since it is publicized by the receiver so that anyone who wishes to send an encrypted message may do so, and the decryption key is called the *private key* since it is kept completely private by the receiver.

The invention of public-key encryption was indeed a revolution in cryptography. It is no coincidence that until the late '70s and early '80s, encryption and cryptography in general belonged to the domain of intelligence and military organizations. It was only with the advent of public-key techniques did the use of cryptography spread to the masses.

A public-key encryption scheme enables private communication without *ever* relying on private channels.[1] A receiver can publicize her public key in the newspaper or on her webpage, thus enabling anyone to send her encrypted messages. The receiver could also simply send her public key via email to a particular sender of interest, without having to worry about the fact that an adversary eavesdropping on all her communication will also observe this public key.[2]

Let us summarize how public-key encryption addresses the limitations of the private-key setting as discussed in Section 9.1:

1. Public-key encryption allows key distribution to be done over public channels. This can potentially simplify initial deployment of the system, and can also ease maintenance of the system when parties join or leave.

2. Public-key encryption vastly reduces the need to store many secret keys. Even if all pairs of parties want the ability to communicate securely, each party need only store his own *private* key in a secure fashion. Other parties' public keys can either be obtained when needed, or stored in a *non-secure* (i.e., publicly-readable) fashion.

3. Finally, public-key cryptography is (more) suitable for open environments where parties who have never previously interacted want the ability to communicate securely. For example, a merchant can post their public key on-line; any user making a purchase can obtain the merchant's public key, as needed, when they need to encrypt their credit card information.

In fairness, we should emphasize that the above discussion glosses over a number of issues, most importantly the need to ensure *authentic* distribution of public keys in the first place. The reader will have to wait until Section 12.8 for a more complete discussion of this point.

Public-key primitives. Diffie and Hellman actually introduced three distinct public-key (or asymmetric) primitives. The first is that of public-key

[1] For now, however, we do assume *authenticated* channels whereby each party "knows" who is communicating on the channel at the other end. In Section 12.8, when we discuss certification authorities and public-key infrastructures, we revisit this assumption.

[2] We assume here that the adversary cannot *replace* her public key. That is, we assume a public but authenticated channel; see the previous footnote.

encryption, described above; we study this notion in Chapters 10 and 11. The second is a public-key analogue of message authentication codes, called *digital signatures* and is introduced in Chapter 12. As with MACs, a digital signature scheme is used to prevent undetected tampering of a message. In contrast to MACs, however, authenticity of a message can be verified by anyone knowing only the public key of the sender. This turns out to have far-reaching ramifications. Specifically, it is possible to take a document that was digitally signed by Alice and present it to a third party, say, a judge, as proof that Alice indeed signed the document. Since only Alice knows the corresponding private key, this serves as proof that Alice signed the document. This property is called *non-repudiation* and has extensive applications in electronic commerce. For example, it is possible to digitally sign contracts, send signed electronic purchase orders or promises of payments and so on. Digital signatures are also used to aid in the secure distribution of public keys within a "public-key infrastructure". This is discussed in more detail in Chapter 12.

The third primitive introduced by Diffie and Hellman is that of *interactive key exchange*. An interactive key-exchange protocol is a method whereby parties who do not share any secret information can generate a shared, secret key by communicating over a public channel. The main property guaranteed here is that an eavesdropping adversary who sees all the messages sent over the communication line does not learn anything about the resulting secret key. Stopping to think about it, the existence of secure key exchange is quite amazing — it means that, in principle, if you and a friend stand on opposite sides of a room you can shout messages to each other in such a way that will allow you to generate a shared secret that someone else (listening to everything you say) cannot learn anything about!

The main difference between key exchange and encryption is that the former is an interactive protocol, and so both parties are required to be on-line simultaneously. In contrast, encryption is (typically) a non-interactive process and is thus more appropriate for some applications. Secure email, for example, requires encryption because the recipient is not necessarily on-line when the email message is sent.

Although Diffie and Hellman introduced all three of the above primitives (i.e., public-key encryption, digital signatures, and key exchange), they only presented a construction of a key-exchange protocol. We describe the Diffie-Hellman key-exchange protocol in the next section. A year later, Ron Rivest, Adi Shamir, and Len Adleman proposed the *RSA problem* and presented the first public-key encryption and digital signature schemes based on the hardness of this problem. Variants of their schemes are now among the most widely used cryptographic schemes today. Interestingly, in 1985, El Gamal presented an encryption scheme that is essentially a slight twist on the Diffie-Hellman key-exchange protocol. Thus, although Diffie and Hellman did not succeed in constructing a (non-interactive) public-key encryption scheme, they came very close.

History. It is fascinating to read about the history leading to the public-key revolution initiated by Diffie and Hellman. Similar ideas were being worked on by others around the same time. Another researcher doing similar and independent work was Ralphe Merkle, considered by many to be a co-inventor of public-key cryptography (though he published after Diffie and Hellman). We mention also the work of Michael Rabin, who developed constructions of signature schemes and public-key encryption schemes based on the hardness of factoring about 1 year after the work of Rivest, Shamir, and Adleman. It appears also that a public-key encryption scheme was known to the intelligence world (at the British intelligence agency GCHQ) in the early 1970s, prior to the publication of the Diffie-Hellman paper. Although the underlying mathematics of public-key encryption may have been discovered before 1976, it is fair to say that the widespread ramifications of this new technology were not appreciated until Diffie and Hellman came along.

At the time their work was carried out, Diffie and Hellman (and others publishing papers in cryptography) were essentially under threat of prosecution. This is due to the fact that under the International Traffic in Arms Regulations (ITAR), technical literature on cryptography was considered an implement of war. Although cryptographic publications soon became accepted and widespread, at the time they were considered by some to be highly sensitive. Hellman tells a story where he personally gave a conference presentation of joint work with Ralph Merkle and Steven Pohlig (who were graduate students at the time) because Stanford's general counsel recommended that students not present the paper lest they be prosecuted. Fortunately, the US government did not pursue this route and publications in cryptography were allowed to continue. (The US still imposes limitations on the export of cryptographic *implementations*. Since 2000, however, these restrictions have been greatly relaxed and are today hardly felt at all.)

9.4 Diffie-Hellman Key Exchange

In this section we present the Diffie-Hellman key-exchange protocol and prove its security in the presence of eavesdropping adversaries. Security against a passive, eavesdropping adversary is a relatively weak guarantee, and we emphasize that in practice security must hold even against active adversaries who may intercept and modify messages sent between the parties. We will not define a notion of security for such adversaries, however, as this material is beyond the scope of this book. (Moreover, we are interested here in the setting where the parties have no shared cryptographic keys to begin with, in which case there is not much that can be done to prevent an adversary from impersonating one of the parties.)

The setting and definition of security. We consider a setting with two parties Alice and Bob who run some protocol in order to generate a shared, secret key; we denote the protocol by Π (thus, Π can be viewed as the set of instructions for Alice and Bob in the protocol). Alice and Bob begin by holding the security parameter 1^n; they then choose (independent) random coins and run the protocol Π. At the end of the protocol, Alice and Bob output keys $k_A, k_B \in \{0,1\}^n$, respectively. The basic correctness requirement we will impose on Π is that it should always hold that $k_A = k_B$ (i.e., for all choices of random coins by Alice and Bob). Since we will only deal with protocols that satisfy this requirement, we will speak simply of *the* key $k = k_A = k_B$ generated by an honest execution of Π.

We now turn to defining security. Intuitively, a key-exchange protocol is secure if the key output by Alice and Bob is completely unknown to an eavesdropping adversary. This is formally defined by requiring that an adversary who has eavesdropped on an execution of the protocol should be unable to distinguish the key k generated by that execution (and now shared by Alice and Bob) from a *completely random* key of length n. This is much stronger than simply requiring that the adversary be unable to *compute* k exactly, and this stronger notion is necessary if the parties will use k to perform some cryptographic task (e.g., to use k within a private-key encryption scheme).

Formalizing the above, Let Π be a key-exchange protocol, $\mathcal{A}$ an adversary, and n the security parameter. We have the following experiment:

The key-exchange experiment $\mathsf{KE}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$:

1. *Two parties holding 1^n execute protocol Π. This execution of the protocol results in a transcript* trans *containing all the messages sent by the parties, and a key k that is output by each of the parties.*
2. *A random bit $b \leftarrow \{0,1\}$ is chosen. If $b = 0$ then choose $\hat{k} \leftarrow \{0,1\}^n$ uniformly at random, and if $b = 1$ set $\hat{k} := k$.*
3. *$\mathcal{A}$ is given* trans *and $\hat{k}$, and outputs a bit b'.*
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. (In case $\mathsf{KE}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$, we say that $\mathcal{A}$* succeeds.*)*

The fact that $\mathcal{A}$ is given the transcript trans reflects the fact that $\mathcal{A}$ eavesdrops on the entire execution of the protocol and thus sees all messages exchanged by the parties. In the real world, of course, $\mathcal{A}$ would not be given any key; in the experiment above the adversary is given $\hat{k}$ only as a means of defining what it means for $\mathcal{A}$ to "break" the security of Π. That is, the adversary succeeds in "breaking" Π if it can correctly determine whether the key $\hat{k}$ it was given is the "correct" key corresponding to the given execution of the protocol, or whether $\hat{k}$ is a completely random key that is independent of the transcript. As expected, we say that Π is secure if the adversary succeeds with probability that is at most negligibly greater than 1/2. That is:

DEFINITION 9.1 *A key-exchange protocol* Π *is* secure in the presence of an eavesdropper *if for every probabilistic polynomial-time adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{KE}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n).$$

The aim of a key-exchange protocol is almost always to generate a shared key k that will be used by the parties for some further cryptographic purpose, e.g., to encrypt and authenticate their subsequent communication using private-key encryption and message authentication codes, respectively. Intuitively, we expect this approach to be secure since the key k "looks like" a random key. Nevertheless, the above definition says nothing about whether such usage of the key will actually achieve the desired security properties and it is always dangerous to rely on intuition alone. Fortunately, it is possible to prove that the above definition suffices for this application; see Exercise 9.1.

The Diffie-Hellman key-exchange protocol. We now describe the key-exchange protocol that appeared in the original paper by Diffie and Hellman (though they were less formal than we will be here). Let $\mathcal{G}$ be a probabilistic polynomial-time algorithm that, on input 1^n, outputs a (description of a) cyclic group $\mathbb{G}$, its order q (with $\|q\| = n$), and a generator g of $\mathbb{G}$. (As usual, we also require that the group operation in $\mathbb{G}$ can be computed in time polynomial in n.) See, e.g., Section 7.3.3 for one possibility for $\mathcal{G}$. The Diffie-Hellman key-exchange protocol, described for two parties Alice and Bob, is given as Construction 9.2 and illustrated in Figure 9.2.

CONSTRUCTION 9.2

- **Common input:** The security parameter 1^n
- **The protocol:**
  1. Alice runs $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$.
  2. Alice chooses $x \leftarrow \mathbb{Z}_q$ uniformly at random, and computes $h_1 := g^x$.
  3. Alice sends $(\mathbb{G}, q, g, h_1)$ to Bob.
  4. Bob receives $(\mathbb{G}, q, g, h_1)$. He chooses $y \leftarrow \mathbb{Z}_q$ uniformly at random and computes $h_2 := g^y$. Bob sends h_2 to Alice and outputs the key $k_B := h_1^y$.
  5. Alice receives h_2 and outputs the key $k_A := h_2^x$.

The Diffie-Hellman key-exchange protocol.

In our description, we have assumed that Alice generates $(\mathbb{G}, q, g)$ and sends these to Bob with her first message. In certain settings — say, when Alice and

Bob are both employees in the same company, or when they both trust some centralized authority — it is possible that both parties have instead agreed upon $(\mathbb{G}, q, g)$ in advance, in which case Alice need only send h_1, and Bob need not wait to receive Alice's message before computing and sending h_2.

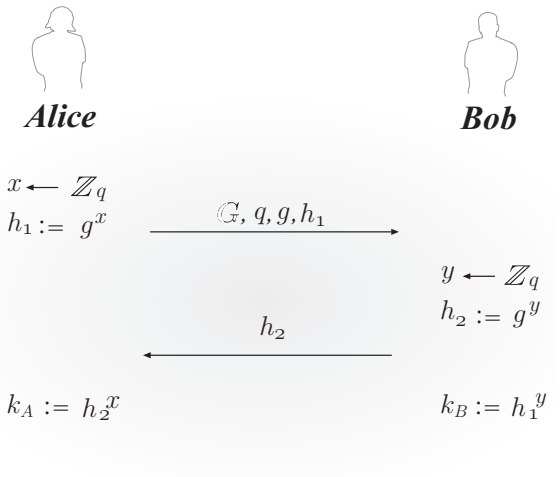

FIGURE 9.2: The Diffie-Hellman key-exchange protocol.

It is not hard to see that the protocol is correct: Bob computes the key

$$k_B = h_1^y = (g^x)^y = g^{xy}$$

and Alice computes the key

$$k_A = h_2^x = (g^y)^x = g^{xy},$$

and so $k_A = k_B$. (The observant reader will note that the shared key is a group element, not a bit-string. We will return to this point later.) We now focus on security of the protocol.

Diffie and Hellman did not prove security of their protocol; indeed, the appropriate notions (both the definitional framework as well as the idea of formulating precise assumptions) were not yet in place. Let us see what sort of assumption will be needed in order for the protocol to be secure. A first observation, made also by Diffie and Hellman, is that a *minimal* requirement for security of the protocol is for the discrete logarithm problem to be hard relative to $\mathcal{G}$. If not, then an adversary given the transcript can compute the secret value of one of the parties and then easily compute the key: i.e., given g and h_1 — and assuming the discrete logarithm problem is easy — an adversary can compute $x := \log_g h_1$ (which is Alice's secret value) and then compute $k_A := h_2^x$ just as Alice does. So, hardness of the discrete logarithm problem is necessary for the protocol to be secure. It is not, however,

sufficient, as is possible that there may be other ways of computing the key $k_A = k_B$ without explicitly finding x or y. The *computational* Diffie-Hellman assumption — which would only guarantee that the key g^{xy} is hard to compute in its entirety — does not suffice either. What is required by Definition 9.1 is exactly that g^{xy} should be *indistinguishable from random* for any adversary given g, g^x, and g^y. This is exactly the *decisional* Diffie-Hellman assumption introduced in Section 7.3.2.

As we will see, a proof of security for the protocol follows almost immediately from this assumption. This should not be surprising, as the Diffie-Hellman assumptions were introduced — well after Diffie and Hellman published their paper — as a way of abstracting the properties underlying the observed security of the Diffie-Hellman key-exchange protocol. Given this, it is fair to ask whether anything is gained by defining and proving security in the first place. By this point in the book, hopefully you are already convinced the answer is *yes*: by precisely defining secure key exchange we are forced to think about exactly what security properties we require (and, consequently, can study what security properties Diffie-Hellman key exchange achieves); by specifying a precise assumption (that is, the decisional Diffie-Hellman assumption) we can study this assumption independently of any particular application; finally, once we have become convinced that the decisional Diffie-Hellman assumption is valid we can construct other protocols relying on the same underlying assumption.

For completeness, we now prove security of the Diffie-Hellman key-exchange protocol based on the decisional Diffie-Hellman assumption. Actually, we are going to "cheat" and not prove it secure with respect to Definition 9.1. In fact, the protocol as described in Construction 9.2 is *not*, in general, secure with respect to this definition. The issue is that Definition 9.1 requires the key output by the parties to be indistinguishable from a *random string*, whereas we will only be able to prove that the output of the parties in Construction 9.2 is indistinguishable from a *random element of* $\mathbb{G}$. (In general, a random group element will look very different from a random string.) This discrepancy needs to be addressed if the protocol is to be used in practice — after all, group elements are not typically very useful as cryptographic keys — and we briefly discuss one standard way to do so following the proof. For now, we let $\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ denote a modified experiment where if $b = 0$ the adversary is given $\hat{k} \leftarrow \mathbb{G}$ chosen uniformly at random, instead of a random string.

THEOREM 9.3 *If the decisional Diffie-Hellman problem is hard relative to $\mathcal{G}$, then the Diffie-Hellman key-exchange protocol Π is secure in the presence of an eavesdropper (with respect to the modified experiment $\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}$).*

PROOF The intuition behind the proof has been given above and we therefore proceed immediately to the details. Let $\mathcal{A}$ be a PPT adversary.

Since $\Pr[b=0] = \Pr[b=1] = 1/2$, we have

$$\Pr\left[\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \tag{9.1}$$
$$= \frac{1}{2} \cdot \Pr\left[\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1 \mid b = 1\right] + \frac{1}{2} \cdot \Pr\left[\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1 \mid b = 0\right].$$

In experiment $\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ the adversary $\mathcal{A}$ receives $(\mathbb{G}, q, g, h_1, h_2, \hat{k})$, where the first set of values $(\mathbb{G}, q, g, h_1, h_2)$ represents the transcript of the protocol execution, and where $\hat{k}$ is either the actual key g^{xy} computed by the parties (if $b = 1$) or a random group element (if $b = 0$). Distinguishing between these two cases is exactly equivalent to solving the decisional Diffie-Hellman problem. That is, using Equation (9.1):

$$\begin{aligned}
&\Pr\left[\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \\
&= \frac{1}{2} \cdot \Pr\left[\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1 \mid b = 1\right] + \frac{1}{2} \cdot \Pr\left[\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1 \mid b = 0\right] \\
&= \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathbb{G}, g, q, g^x, g^y, g^{xy}) = 1] + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 0] \\
&= \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathbb{G}, g, q, g^x, g^y, g^{xy}) = 1] + \frac{1}{2} \cdot \big(1 - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1]\big) \\
&= \frac{1}{2} + \frac{1}{2} \cdot \big(\Pr[\mathcal{A}(\mathbb{G}, g, q, g^x, g^y, g^{xy}) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1]\big) \\
&\leq \frac{1}{2} + \frac{1}{2} \cdot \big|\Pr[\mathcal{A}(\mathbb{G}, g, q, g^x, g^y, g^{xy}) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1]\big|,
\end{aligned}$$

where the probabilities in the final three lines are taken over $(\mathbb{G}, q, g)$ output by $\mathcal{G}(1^n)$, and $x, y, z \leftarrow \mathbb{Z}_q$ chosen uniformly at random. (Note that, since g is a generator, g^z is a uniformly distributed element of $\mathbb{G}$ when z is uniformly distributed in $\mathbb{Z}_q$.) If the decisional Diffie-Hellman assumption is hard relative to $\mathcal{G}$, that exactly means that there exists a negligible function negl for which

$$\big|\Pr[\mathcal{A}(\mathbb{G}, g, q, g^x, g^y, g^{xy}) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1]\big| \leq \mathsf{negl}(n).$$

We conclude that

$$\Pr\left[\widehat{\mathsf{KE}}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \frac{1}{2} \cdot \mathsf{negl}(n),$$

completing the proof. ∎

Random group elements vs. random strings. The previous theorem shows that the key output by Alice and Bob is computationally indistinguishable from a random group element (for a polynomial-time eavesdropper). For this key to be useful for subsequent cryptographic operations, however, it should be indistinguishable from a random string of some appropriate length.

This can be achieved by mapping group elements to strings in some way that "preserves uniformity". Efficient techniques for doing so exist, and these are called "randomness extractors". These are by now relatively standard in computer science, but are beyond the scope of this book.

Active adversaries. So far we have considered only the case of an eavesdropping adversary. Although eavesdropping attacks are by far the most common (as they are so easy to carry out), they are by no means the only possible attack. *Active* attacks, in which the adversary sends messages of its own to one or both of the parties are also a concern, and any protocol used in practice must be resilient to active attacks as well. When considering active attacks, it is useful to distinguish, informally, between *impersonation* attacks where only one of the honest parties is executing the protocol and the adversary impersonates the other party, and *man-in-the-middle* attacks where both honest parties are executing the protocol and the adversary is intercepting and modifying messages being sent from one party to the other.

We will not define security against either class of attacks, as such a definition is rather involved and also cannot be achieved without the parties sharing *some* information in advance. Nevertheless, it is worth remarking that the Diffie-Hellman protocol is *completely insecure* against man-in-the-middle attacks. In fact, a man-in-the-middle adversary can act in such a way that Alice and Bob terminate the protocol with different keys k_A and k_B that are both known to the adversary, yet neither Alice nor Bob can detect that any attack was carried out. We leave the details of this attack as an exercise.

The fact that the Diffie-Hellman protocol is not resilient to man-in-the-middle attacks does not detract in any way from its importance. The Diffie-Hellman protocol served as the first demonstration that asymmetric techniques (and number-theoretic problems) could be used to alleviate the problems of key distribution in cryptography. Furthermore, extensions of the Diffie-Hellman protocol can be shown to prevent man-in-the-middle attacks, and such protocols are widely used today.

The Diffie-Hellman key-exchange protocol in practice. The Diffie-Hellman protocol in its basic form is typically not used in practice due to its insecurity against man-in-the-middle attacks, as discussed above. However, it does form the nucleus of other key-exchange protocols that are resilient to man-in-the-middle attacks and are in wide use today. One notable example of a standardized protocol that relies on Diffie-Hellman key exchange is IPsec.

References and Additional Reading

We have only briefly discussed the problems of key distribution and key management in general. For more information, we recommend looking at

textbooks on network security. Our favorite is the one by Kaufman et al. [87], which provides an excellent treatment of different protocols for secure key distribution, what they aim to achieve, and how they work.

We highly recommend reading the original paper by Diffie and Hellman [47]. The history of the development of public-key cryptography is a fascinating one; the book by Levy [95] focuses on the political and historical aspects of the public-key revolution.

Exercises

9.1 Consider the following *interactive* protocol Π' for encrypting a message: first, the sender and receiver run a key-exchange protocol Π to generate a shared key k. Next, the sender computes $c \leftarrow \mathsf{Enc}_k(m)$ and sends c to the other party, who can decrypt and recover m using k.

(a) Formulate a definition of indistinguishable encryptions in the presence of an eavesdropper (cf. Definition 3.8) appropriate for this interactive setting.

(b) Prove that if Π is secure in the presence of an eavesdropper and $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper, then Π' satisfies your definition given previously.

9.2 Describe in detail a man-in-the-middle attack on the Diffie-Hellman key-exchange protocol whereby the adversary ends up sharing a key k_A with Alice and a (different) key k_B with Bob, and Alice and Bob cannot detect that anything has gone wrong.

What happens if Alice and Bob try to detect the presence of a man-in-the-middle adversary by sending each other (encrypted) questions that only the other party would know how to answer?

9.3 Consider the following key-exchange protocol:

(a) Alice chooses $k, r \leftarrow \{0,1\}^n$ at random, and sends $s := k \oplus r$ to Bob.

(b) Bob chooses $t \leftarrow \{0,1\}^n$ at random and sends $u := s \oplus t$ to Alice.

(c) Alice computes $w := u \oplus r$ and sends w to Bob.

(d) Alice outputs k and Bob computes $w \oplus t$.

Show that Alice and Bob output the same key. Analyze the security of the scheme (i.e., either prove its security or show a concrete attack).

Chapter 10

Public-Key Encryption

10.1 Public-Key Encryption – An Overview

As discussed in the previous chapter, the introduction of public-key encryption marked a revolution in the field of cryptography. Until that time, cryptographers had relied exclusively on shared, *secret* keys to achieve private communication. Public-key techniques, in contrast, enable parties to communicate privately without having agreed on *any* secret information in advance. As noted previously (in a slightly different context), it is quite amazing and counter-intuitive that this is possible: it means that two people on opposite sides of a room who can only communicate by shouting to each other, and have no initial secret, can talk in such a way that no one else in the room learns anything about what they are saying!

In the setting of private-key encryption, two parties agree on a secret key k which can be used (by either party) for both encryption and decryption. Public-key encryption is *asymmetric* in both these respects. Specifically, one party (the *receiver*) generates a *pair* of keys (pk, sk), called the *public key* and the *private key*, respectively. The public key is used by a *sender* to encrypt a message for the receiver; the receiver then uses the private key to decrypt the resulting ciphertext.

Since the goal is to avoid the need for two parties to meet in advance to agree on any information, how does the sender learn pk? At an abstract level, there are essentially two ways this can occur. Let us call the receiver Alice and the sender Bob. If Alice knows that Bob wants to communicate with her, she can at that point generate (pk, sk) (assuming she hasn't done so already) and then send pk *in the clear* to Bob; Bob can then use pk to encrypt his message. We emphasize that the channel between Alice to Bob may be public, but is assumed to be authenticated, meaning that the adversary cannot modify the key sent by Alice to Bob (and, in particular, cannot replace it with its own key). It is possible to use digital signatures to alleviate this problem; see Section 12.8 for a discussion of how public keys can be distributed over unauthenticated channels.

An alternative way to picture the situation is that Alice generates her keys (pk, sk) in advance, *independent* of any particular sender. (In fact, at the time of key generation Alice need not even be aware that Bob wants to talk

to her or even that Bob exists!) Then Alice widely disseminates her public key pk, say, by publishing it on her webpage, putting it on her business cards, publishing it in a newspaper, or placing it in a public directory. Now, *anyone* who wishes to communicate privately with Alice can look up her public key and proceed as above. Note that multiple senders can communicate multiple times with Alice using the same public key pk for all communication.

An important point is that pk is inherently public — and, more to the point, can easily be learned by an attacker — in either of the above scenarios. In the first case, an adversary eavesdropping on the communication between Alice and Bob obtains pk by simply listening to the first message that Alice sends Bob; in the second case, an adversary could just as well look up Alice's public key on his own. A consequence is that the security of public-key encryption cannot rely on the secrecy of pk, but must instead rely on the secrecy of sk. It is therefore crucial that Alice not reveal her private key to anyone, including the sender Bob.

Comparison to Private-Key Encryption

Perhaps the most obvious difference between private- and public-key encryption is that the former assumes *complete* secrecy of all cryptographic keys, whereas the latter requires secrecy for "only" half the key-pair (pk, sk). Although this might seem like a minor distinction, the ramifications are huge: in the private-key setting the communicating parties must somehow be able to share the secret key without allowing any third party to learn it; in the public-key setting, the public key can be sent from one party to the other over a public channel without compromising security. For parties shouting across a room or, more realistically, communicating entirely over a public network like a phone line or the Internet, public-key encryption is the only option.

Another important distinction is that private-key encryption schemes use the same key for both encryption and decryption, while public-key encryption schemes use different keys for each operation. For this reason, public-key encryption schemes are sometimes called *asymmetric*. This asymmetry in the public-key setting means that the roles of sender and receiver are *not* interchangeable the way they are in the private-key setting: a given instance of a public-key encryption scheme allows communication in one direction only. (This can be addressed in any of a number of ways, but the point is that a single invocation of a public-key encryption scheme forces a distinction between one user who acts as a receiver and other users who act as senders.) On the other hand, a single instance of a public-key encryption scheme enables multiple senders to communicate privately with a single receiver, in contrast to the private-key case where a secret key shared between two parties enables only those two parties to communicate privately.

Summarizing and elaborating the preceding discussion, we see that public-key encryption has the following advantages relative to private-key encryption (see also the extensive discussion in the previous chapter):

- The most important advantage is that public-key encryption addresses (to some extent) the *key distribution problem* since communicating parties do not need to secretly share a key in advance of their communication. Public-key encryption allows two parties to communicate secretly even if *all* communication between them is monitored.

- In the case that one receiver is communicating with U senders (e.g., an on-line merchant processing credit card orders from multiple purchasers), it is much more convenient for the receiver to store a *single* private key sk rather than to share, store, and manage U different secret keys (i.e., one for each sender). In fact, when using public-key encryption the number and identities of the potential senders need not be known at the time of key-generation. This allows enormous flexibility, and is clearly essential for an on-line merchant.

The main disadvantage of public-key encryption is that it is at least 2 to 3 *orders of magnitude* slower than private-key encryption.[1] This means, for example, that it can be a challenge to implement public-key encryption in severely resource-constrained devices like smartcards or radio-frequency identification (RFID) tags. Even when a powerful computer is performing cryptographic operations, carrying out many hundreds of such operations may be prohibitive (this is actually quite common as in the case of a server processing credit-card transactions for an on-line merchant). In any case, we may conclude that *if* private-key encryption is an option (i.e., if two parties *can* securely share a key in advance) it should always be used.

In fact, private-key encryption is used *in the public-key setting* to improve the efficiency of the (public-key) encryption of long messages; this is discussed further in Section 10.3. A thorough understanding of private-key encryption is therefore crucial to fully appreciate how public-key encryption is implemented in practice.

Secure Distribution of Public Keys

In our entire discussion thus far, we have implicitly assumed that the adversary is *passive*; that is, the adversary only eavesdrops on communication between the sender and receiver but does not *actively* interfere with the communication. If the adversary has the ability to tamper with *all* communication between the honest parties, and these honest parties hold no shared keys, then privacy simply cannot be achieved. For example, if a receiver Alice sends her public key pk to Bob but the adversary replaces this public key with a key pk' of his own (for which it knows the matching private key sk'), then even though Bob encrypts his message using pk' the adversary will easily be able

[1] It is difficult to give an exact comparison since the relative efficiency depends on the exact schemes under consideration as well as various implementation details.

to recover this message (using sk'). A similar attack works if an adversary is able to change the value of Alice's public key that is stored in some public directory, or if the adversary can tamper with the public key as it is transmitted from the directory to Bob. If Alice and Bob do not share any information in advance, or do not rely on some mutually-trusted third party, there is nothing Alice or Bob can do to prevent active attacks of this sort, or even to tell that such an attack is taking place.[2] Looking ahead, we will discuss in Section 12.8 how mild reliance on a trusted third party can be used to address such attacks.

Our treatment of public-key encryption in this and the next chapter will simply *assume that senders have a legitimate copy of the receiver's public key.* (This will be implicit in the security definitions we provide.) That is, we assume *secure key distribution.* This assumption is made not because active attacks of the type discussed above are of no concern — in fact, they represent a serious threat that must be dealt with in any real-world system that uses public-key encryption. Rather, this assumption is made because there exist other mechanisms for preventing active attacks (see, for example, Section 12.8), and it is therefore convenient (and useful) to decouple the study of secure public-key encryption from the study of secure key distribution.

10.2 Definitions

We begin by defining the syntax of public-key encryption. The definition is very similar to Definition 3.7, with the exception that instead of working with a single key, distinct encryption and decryption keys are defined.

DEFINITION 10.1 *A* public-key encryption scheme *is a tuple of probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *such that:*

1. *The* key generation algorithm Gen *takes as input the security parameter* 1^n *and outputs a pair of keys* (pk, sk)*. We refer to the first of these as the* public key *and the second as the* private key*. We assume for convenience that* pk *and* sk *each have length at least* n*, and that* n *can be determined from* pk, sk*.*

2. *The* encryption algorithm Enc *takes as input a public key* pk *and a message* m *from some underlying plaintext space (that may depend on* pk*). It outputs a ciphertext* c*, and we write this as* $c \leftarrow \mathsf{Enc}_{pk}(m)$*.*

[2] In our "shouting-across-a-room" scenario, Alice and Bob can detect when an adversary interferes with the communication. But this is only because: (1) the adversary cannot prevent Alice's messages from reaching Bob, and (2) Alice and Bob "share" in advance certain information (e.g., the sound of their voices or the way they look) that allows them to "authenticate" their communication.

3. *The* decryption algorithm Dec *takes as input a private key sk and a ciphertext c, and outputs a message m or a special symbol $\perp$ denoting failure. We assume without loss of generality that* Dec *is deterministic, and write this as* $m := \mathsf{Dec}_{sk}(c)$.

It is required that

$$\Pr\left[\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m)) = m\right]$$

except with possibly negligible probability over (sk, pk) output by $\mathsf{Gen}(1^n)$ *and any randomness used by* Enc.

In terms of the syntax of the definition, the important distinction from the private-key setting is that the key-generation algorithm Gen now outputs *two* keys rather than one. (Moreover, we can no longer simply assume that pk is just a random n-bit string.) The public key pk is used for encryption, while the private key sk is used for decryption. Reiterating our earlier discussion, pk is assumed to be widely distributed so that anyone can encrypt messages for the party who has generated this key, but sk must be kept private by the receiver in order for security to possibly hold.

Note that we allow a negligible decryption error and, indeed, the concrete schemes that we will present can have a negligible error (e.g., if a prime needs to be chosen but with negligible probability a composite is obtained instead). Despite this, we will typically ignore this issue from here on.

For practical usage of public-key encryption, we will want the plaintext space to be $\{0,1\}^n$ or $\{0,1\}^*$ (and, in particular, to be independent of the public key). Although we will sometimes describe encryption schemes using some message space $\mathcal{M}$ that does not contain all bit-strings of some fixed length (and that may also depend on the public key), we will in such cases also specify how to encode bit-strings as elements of $\mathcal{M}$. This encoding must be both efficient and efficiently reversible, so the receiver can recover the bit-string that was encrypted.

Example 10.2
Say an encryption scheme has message space $\mathbb{Z}_N$, where N is an n-bit integer that is included in the public key. We can encode strings of length $n-1$ as elements of $\mathbb{Z}_N$ in the natural way, by interpreting any such string as an integer strictly less than N. This encoding is efficient and easily reversible. $\diamondsuit$

10.2.1 Security against Chosen-Plaintext Attacks

We begin our treatment of security notions by introducing the "natural" counterpart of Definition 3.8 in the public-key setting. Since extensive motivation for this definition (as well as the others we will see) has already been

given in Chapter 3, the discussion here will be relatively brief and will focus primarily on the differences between the private-key and public-key settings.

Given a public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and an adversary $\mathcal{A}$, consider the following experiment:

The eavesdropping indistinguishability experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$:

1. $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
2. *Adversary* $\mathcal{A}$ *is given* pk*, and outputs a pair of messages* m_0, m_1 *of the same length. (These messages must be in the plaintext space associated with* pk*.)*
3. *A random bit* $b \leftarrow \{0,1\}$ *is chosen, and then a ciphertext* $c \leftarrow \mathsf{Enc}_{pk}(m_b)$ *is computed and given to* $\mathcal{A}$*. We call* c *the* challenge ciphertext.
4. $\mathcal{A}$ *outputs a bit* b'.
5. *The output of the experiment is defined to be 1 if* $b' = b$*, and 0 otherwise.*

DEFINITION 10.3 *A public-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable encryptions in the presence of an eavesdropper *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

The main difference between the above definition and Definition 3.8 is that here $\mathcal{A}$ *is given the public key* pk. Furthermore, we allow $\mathcal{A}$ to choose its messages m_0 and m_1 based on this public key. This is essential when defining security of public-key encryption since, as discussed previously, we are forced to assume that an adversary eavesdropping on the communication between two parties in the public-key setting knows the public key of the recipient.

The seemingly "minor" modification of giving the adversary $\mathcal{A}$ the public key pk being used to encrypt the message has a tremendous impact: it effectively gives $\mathcal{A}$ access to an encryption oracle *for free*. (The concept of an encryption oracle is explained in Section 3.5.) This is the case because the adversary, given pk, can now encrypt any message m on its own simply by computing $\mathsf{Enc}_{pk}(m)$ using honestly-generated random coins. (As always, $\mathcal{A}$ is assumed to know the algorithm Enc.) The upshot is that Definition 10.3 is *equivalent* to security against chosen-plaintext attacks, defined in a manner analogous to Definition 3.21. Specifically, consider the following experiment defined for public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and adversary $\mathcal{A}$:

The CPA indistinguishability experiment $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$**:**

1. $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
2. *Adversary* $\mathcal{A}$ *is given* pk *as well as oracle access to* $\mathsf{Enc}_{pk}(\cdot)$. *The adversary outputs a pair of messages* m_0, m_1 *of the same length. (These messages must be in the plaintext space associated with* pk.*)*
3. *A random bit* $b \leftarrow \{0,1\}$ *is chosen, and then a ciphertext* $c \leftarrow \mathsf{Enc}_{pk}(m_b)$ *is computed and given to* $\mathcal{A}$. *We call* c *the* challenge ciphertext.
4. $\mathcal{A}$ *continues to have access to* $\mathsf{Enc}_{pk}(\cdot)$*, and outputs a bit* b'.
5. *The output of the experiment is defined to be 1 if* $b' = b$*, and 0 otherwise.*

DEFINITION 10.4 *A public-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable encryptions under a chosen-plaintext attack *(or is* CPA secure*) if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

Summarizing what we have said above, the encryption oracle is unnecessary since $\mathcal{A}$ can encrypt messages by itself using pk. Thus:

PROPOSITION 10.5 *If a public-key encryption scheme* Π *has indistinguishable encryptions in the presence of an eavesdropper, then* Π *also has indistinguishable encryptions under a chosen-plaintext attack.*

This is in contrast to the private-key setting, where there exist schemes that have indistinguishable encryptions in the presence of an eavesdropper but are insecure under a chosen-plaintext attack (see Propositions 3.19 and 3.22). Further differences from the private-key setting that follow almost immediately as consequences of the above are discussed next.

Impossibility of Perfectly-Secret Public-Key Encryption

Perfectly-secret public-key encryption could be defined analogously to Definition 2.1 by conditioning over the entire view of an eavesdropper (i.e., including the public key). Equivalently, it could be defined by extending Definition 10.3 to require that for *all* adversaries $\mathcal{A}$

$$\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \frac{1}{2}.$$

In contrast to the private-key setting, perfectly-secret public-key encryption is impossible, regardless of how long the keys are and how short the message is. In fact, given pk and a ciphertext c computed via $c \leftarrow \mathsf{Enc}_{pk}(m)$, it is possible for an unbounded adversary to determine the message m with probability 1. A demonstration of this is left as an exercise.

Insecurity of Deterministic Public-Key Encryption

As noted in the context of private-key encryption, no deterministic encryption scheme can be CPA-secure. Due to the equivalence between CPA-security and indistinguishability of encryptions in the presence of an eavesdropper in the public-key setting, we conclude that:

THEOREM 10.6 *No deterministic public-key encryption scheme has indistinguishable encryptions in the presence of an eavesdropper.*

Because Theorem 10.6 is so important, it merits a bit more discussion. The theorem is *not* a mere "artifact" of our security definition, or an indication that Definition 10.3 is too strong. Deterministic public-key encryption schemes are vulnerable to *practical* attacks in *realistic* scenarios, and should never be used. The reason is that a deterministic scheme not only allows the adversary to determine when the same message is sent twice (as in the private-key setting), but also allows the adversary to recover the message, with probability 1, as long as the set of possible messages being encrypted is small. (See Exercise 10.2.) For example, consider a professor encrypting the final grade of a student. Here, an eavesdropper knows that the student's grade must be one of $\{A, B, C, D, F\}$. If the professor uses a deterministic public-key encryption scheme, an eavesdropper can quickly determine the student's actual grade by encrypting all possible grades and comparing the result to the given ciphertext.

Although the above theorem seems deceptively simple, for a long time *many real-world systems were designed using deterministic public-key encryption.* When public-key encryption was introduced it is fair to say that the importance of probabilistic encryption was not yet fully realized. The seminal work of Goldwasser and Micali, in which (something equivalent to) Definition 10.3 was proposed and Theorem 10.6 was proved, marked a turning point in the field of cryptography. The importance of pinning down one's intuition in a formal definition, and looking at things the right way for the first time — even if seemingly simple in retrospect — should not be underestimated.

10.2.2 Multiple Encryptions

As in Chapter 3, it is important to understand the effect of using the same key (in this case, the same public key) for encrypting multiple messages.

We define security in such a setting via an extension of the definition of eavesdropping security (Definition 10.3), though it should be clear from the discussion in the previous section that the definition we give is automatically equivalent to a definition in which chosen-plaintext attacks are also allowed. We then prove that any scheme having indistinguishable encryptions in the presence of an eavesdropper is automatically secure even when used to encrypt multiple messages. This means that we can prove security with respect to the former definition, which is simpler and easier to work with, and conclude that the scheme satisfies the latter definition, which more accurately models adversarial attacks.

An analogous result in the private-key setting was stated, but not proved, as Proposition 3.22. That claim refers to security under a chosen-plaintext attack, but as we have seen this is equivalent to eavesdropping in this setting.

Consider the following experiment defined for a public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and adversary $\mathcal{A}$:

The multiple message eavesdropping experiment $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$:

1. $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
2. *Adversary* $\mathcal{A}$ *is given* pk*, and outputs a pair of vectors of messages* $\vec{M}_0 = (m_0^1, \ldots, m_0^t)$ *and* $\vec{M}_1 = (m_1^1, \ldots, m_1^t)$ *such that* $|m_0^i| = |m_1^i|$ *for all* i*. (All messages must be in the plaintext space associated with* pk*.)*
3. *A random bit* $b \in \{0,1\}$ *is chosen. For all* i*, the ciphertext* $c^i \leftarrow \mathsf{Enc}_{pk}(m_b^i)$ *is computed and the vector of ciphertexts* $\vec{C} = (c^1, \ldots, c^t)$ *is given to* $\mathcal{A}$*. Adversary* $\mathcal{A}$ *then outputs a bit* b'.
4. *The output of the experiment is defined to be 1 if* $b' = b$*, and 0 otherwise.*

DEFINITION 10.7 *A public-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable multiple encryptions in the presence of an eavesdropper *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

The proof that the above definition is equivalent to Definition 10.3 is not difficult, though it is a bit technical. We therefore provide some intuition before giving the formal proof. For this discussion we deal with the case that $t = 2$ in $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$. (In general, t can be arbitrary and may even depend on n or pk.) Fix an arbitrary PPT adversary $\mathcal{A}$ and a public-key encryption scheme Π that has indistinguishable encryptions in the presence of an eavesdropper, and consider experiment $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$ with $t = 2$. In this

experiment, if $b = 0$ the adversary is given $\vec{C} = (\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2))$ while if $b = 1$ the adversary is given $\vec{C} = (\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2))$.

We show that there exists a negligible function negl such that

$$\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

We use the fact that

$$\begin{aligned}\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] = \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\ + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1],\end{aligned}$$

where the equality follows by conditioning on the two possible values of b.

Consider what would happen if $\mathcal{A}$ were given $(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2))$; i.e., a pair of ciphertexts where the first is an encryption from $\vec{M}_0$ and the second is an encryption from $\vec{M}_1$. Although this does not correspond to anything that can happen in experiment $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$, the probability that $\mathcal{A}$ outputs 0 when given two ciphertexts computed in this way is still well-defined. We claim:

CLAIM 10.8 *There exists a negligible function* negl *such that*

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}(n) \geq \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\ + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 1].\end{aligned}$$

PROOF To prove the claim, construct the following PPT adversary $\mathcal{A}'$ that eavesdrops on the encryption of a *single* message.

Adversary $\mathcal{A}'$:

1. $\mathcal{A}'$, given pk, runs $\mathcal{A}(pk)$ to obtain two vectors of messages $\vec{M}_0 = (m_0^1, m_0^2)$ and $\vec{M}_1 = (m_1^1, m_1^2)$.
2. $\mathcal{A}'$ outputs a pair of messages (m_0^2, m_1^2), and is given in return a ciphertext c^2.
3. $\mathcal{A}'$ computes $c^1 \leftarrow \mathsf{Enc}_{pk}(m_0^1)$; runs $\mathcal{A}(c^1, c^2)$; and outputs the bit b' that is output by $\mathcal{A}$.

If we look at the experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n)$, we see that when $b = 0$ adversary $\mathcal{A}'$ is given $\mathsf{Enc}_{pk}(m_0^2)$. Furthermore,

$$\Pr[\mathcal{A}'(\mathsf{Enc}_{pk}(m_0^2)) = 0] = \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0].$$

In contrast, when $b = 1$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n)$, adversary $\mathcal{A}'$ is given $\mathsf{Enc}_{pk}(m_1^2)$. Furthermore,

$$\Pr[\mathcal{A}'(\mathsf{Enc}_{pk}(m_1^2)) = 1] = \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 1].$$

By the security of Π (in the sense of single-message indistinguishability), there exists a negligible function negl such that

$$\begin{aligned}
\frac{1}{2} + \mathsf{negl}(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1] \\
&= \frac{1}{2} \cdot \Pr[\mathcal{A}'(\mathsf{Enc}_{pk}(m_0^2)) = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}'(\mathsf{Enc}_{pk}(m_1^2)) = 1] \\
&= \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\
&\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 1],
\end{aligned}$$

completing the proof of the claim. ■

By a very similar argument, one can also prove:

CLAIM 10.9 *There exists a negligible function* negl *such that*

$$\begin{aligned}
\frac{1}{2} + \mathsf{negl}(n) \geq \frac{1}{2} &\cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 0] \\
&+ \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1].
\end{aligned}$$

Summing the expressions in these two claims and using the fact that the sum of two negligible functions is negligible, we see that there exists a negligible function negl such that:

$$\begin{aligned}
1 + \mathsf{negl}(n) &\geq \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\
&\quad + \frac{1}{2} \cdot \Big(\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 1] \\
&\qquad + \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 0] \Big) \\
&\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1] \\
&= \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] + \frac{1}{2} \\
&\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1] \\
&= \frac{1}{2} + \Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1],
\end{aligned}$$

implying that

$$\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$$

as desired. This completes the proof of equivalence for the case $t = 2$.

The main complication that arises in the general case is that t is no longer fixed but may instead depend on n. The formal proof of the theorem that follows serves as a good illustration of the *hybrid argument* which is used extensively in the analysis of more complex cryptographic schemes; on a first reading, though, the reader may want to skip the proof.

THEOREM 10.10 *If a public-key encryption scheme* Π *has indistinguishable encryptions in the presence of an eavesdropper, then* Π *has indistinguishable* multiple *encryptions in the presence of an eavesdropper.*

PROOF Fix an arbitrary PPT adversary $\mathcal{A}$, and consider experiment $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$. Let $t = t(n)$ be an upper-bound on the number of messages in each of the two vectors output by $\mathcal{A}$, and assume without loss of generality that $\mathcal{A}$ always outputs vectors containing *exactly* this many messages. (That this is indeed without loss of generality is left as an exercise.) For a given public key pk and vectors $\vec{M}_0 = (m_0^1, \ldots, m_0^t)$ and $\vec{M}_1 = (m_1^1, \ldots, m_1^t)$ output by $\mathcal{A}$, define

$$\vec{C}^{(i)} \stackrel{\text{def}}{=} (\underbrace{\mathsf{Enc}_{pk}(m_0^1), \ldots, \mathsf{Enc}_{pk}(m_0^i)}_{i \text{ terms}}, \underbrace{\mathsf{Enc}_{pk}(m_1^{i+1}), \ldots, \mathsf{Enc}_{pk}(m_1^t)}_{t-i \text{ terms}})$$

for $0 \leq i \leq t$. We stress that the above encryptions are always performed using independent random coins, and so the above actually represents a *distribution* over vectors containing t ciphertexts. Using this notation, we have

$$\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] = \frac{1}{2} \cdot \Pr[\mathcal{A}(\vec{C}^{(t)}) = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}(\vec{C}^{(0)}) = 1]. \quad (10.1)$$

Consider the following PPT adversary $\mathcal{A}'$ that eavesdrops on the encryption of a *single* message.

Adversary $\mathcal{A}'$:

1. $\mathcal{A}'$, given pk, runs $\mathcal{A}(pk)$ to obtain two vectors of messages $\vec{M}_0 = (m_0^1, \ldots, m_0^t)$ and $\vec{M}_1 = (m_1^1, \ldots, m_1^t)$, with $t = t(n)$.
2. $\mathcal{A}'$ chooses a random index $i \leftarrow \{1, \ldots, t\}$ and outputs the pair of messages m_0^i, m_1^i. $\mathcal{A}'$ is given in return a ciphertext c^i.
3. For $j < i$, $\mathcal{A}'$ computes $c^j \leftarrow \mathsf{Enc}_{pk}(m_0^j)$. For $j > i$, $\mathcal{A}'$ computes $c^j \leftarrow \mathsf{Enc}_{pk}(m_1^j)$. Then $\mathcal{A}'$ runs $\mathcal{A}(c^1, \ldots, c^t)$ and outputs the bit b' that is output by $\mathcal{A}$.

Intuitively, we view $\mathcal{A}'$ as "guessing" an index $i \in \{1, \ldots, t\}$ for which $\mathcal{A}$ distinguishes between $\vec{C}^{(i)}$ and $\vec{C}^{(i-1)}$. This is because the ith element in the vector $(c^1, \ldots, c^t)$ given to $\mathcal{A}$ is c^i, the challenge ciphertext of $\mathcal{A}'$. Thus, if $\mathcal{A}$ distinguishes between $\vec{C}^{(i)}$ and $\vec{C}^{(i-1)}$ this means that it actually distinguishes between an encryption of m_0^i and an encryption of m_1^i (because this is the only difference between $\vec{C}^{(i)}$ and $\vec{C}^{(i-1)}$). This is the core idea of the proof below.

In experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n)$, when $b = 0$ and $i = i^*$, adversary $\mathcal{A}'$ is given $\mathsf{Enc}_{pk}(m_0^{i^*})$ and $\mathcal{A}$ is run on input distributed according to $\vec{C}^{(i^*)}$. So,

$$\begin{aligned}\Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] &= \sum_{i^*=1}^{t} \Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0 \wedge i = i^*] \cdot \Pr[i = i^*] \\ &= \sum_{i^*=1}^{t} \frac{1}{t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 0].\end{aligned}$$

On the other hand, when $b = 1$ and $i = i^*$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n)$, adversary $\mathcal{A}'$ is given $\mathsf{Enc}_{pk}(m_1^{i^*})$ and $\mathcal{A}$ is run on input distributed according to $\vec{C}^{(i^*-1)}$. So,

$$\begin{aligned}\Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1] &= \sum_{i^*=1}^{t} \Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1 \wedge i = i^*] \cdot \Pr[i = i^*] \\ &= \sum_{i^*=1}^{t} \frac{1}{t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*-1)}) = 1] \\ &= \sum_{i^*=0}^{t-1} \frac{1}{t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 1],\end{aligned}$$

(where the third equality is just by shifting the indices of the summation).

By assumption, Π has indistinguishable encryptions in the presence of an eavesdropper, and so there exists a negligible function negl such that:

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1] \\ &= \frac{1}{2}\Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] + \frac{1}{2}\Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1] \\ &= \sum_{i^*=1}^{t} \frac{1}{2t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 0] + \sum_{i^*=0}^{t-1} \frac{1}{2t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 1] \\ &= \frac{1}{2t} \cdot \sum_{i^*=1}^{t-1} \Big(\Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 0] + \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 1] \Big) \\ &\quad + \frac{1}{2t} \cdot \Big(\Pr[\mathcal{A}(\vec{C}^{(t)}) = 0] + \Pr[\mathcal{A}(\vec{C}^{(0)}) = 1] \Big) \\ &= \frac{t-1}{2t} + \frac{1}{t} \cdot \Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1],\end{aligned}$$

where the last equality uses Equation (10.1). Simple algebra shows that this implies

$$\frac{1}{2} + t(n) \cdot \mathsf{negl}(n) \geq \Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1].$$

Because t is polynomial, the function $t(n) \cdot \mathsf{negl}(n)$ is also negligible. Since $\mathcal{A}$ is an arbitrary PPT adversary, this completes the proof that Π has indistinguishable multiple encryptions in the presence of an eavesdropper. ■

Encrypting Arbitrary-Length Messages

An immediate consequence of the above result is that given any public-key encryption scheme for *fixed-length messages* that has indistinguishable encryptions in the presence of an eavesdropper, we can obtain a public-key encryption scheme for *arbitrary-length messages* satisfying the same notion of security. We illustrate this in the extreme case when the original scheme encrypts only 1-bit messages. Say $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is an encryption scheme where the plaintext space is $\{0,1\}$. We can construct a new scheme $\Pi' = (\mathsf{Gen}, \mathsf{Enc}', \mathsf{Dec}')$ with plaintext space $\{0,1\}^*$ by defining Enc' as follows:

$$\mathsf{Enc}'_{pk}(m) = \mathsf{Enc}_{pk}(m_1), \ldots, \mathsf{Enc}_{pk}(m_t), \tag{10.2}$$

where $m = m_1 \cdots m_t$. The decryption algorithm Dec' is modified in the obvious way. We have:

PROPOSITION 10.11 *Let Π and Π' be as above. If Π has indistinguishable encryptions in the presence of an eavesdropper, then so does Π'.*

Proposition 10.11 is true when Π has indistinguishable encryptions in the presence of an eavesdropper (and, by extension, when Π is CPA-secure), but an analogous result is *not* true for the case of security against chosen-ciphertext attacks. See Section 10.6 and Exercise 10.13.

A note on terminology. We have shown that in the *public-key setting* any scheme having indistinguishable encryptions in the presence of an eavesdropper is also CPA-secure. This means that once we prove a scheme secure with respect to the former (relatively weak) definition we obtain as an immediate consequence security with respect to the latter (more realistic) definition. We will therefore refer to schemes as being "CPA-secure" in our discussion and theorem statements, but in our proofs we will work exclusively with the notion of indistinguishable encryptions in the presence of an eavesdropper.

10.3 Hybrid Encryption

Proposition 10.11 shows that any CPA-secure public-key encryption scheme for 1-bit messages can be used to obtain a CPA-secure encryption scheme for messages of arbitrarily length. Encrypting a t-bit message using this approach requires t invocations of the original encryption scheme, meaning that both the computation and the ciphertext length are increased by a multiplicative factor of t relative to the underlying scheme. (Of course, if the original scheme can encrypt strings of length n, then only $\lceil t/n \rceil$ invocations are needed. However, for long messages this is still unreasonable.)

It is possible to do significantly better, for messages that are sufficiently long, by using private-key encryption *in tandem with* public-key encryption. This improves efficiency because private-key encryption is significantly more efficient than public-key encryption. The resulting combination is called *hybrid encryption* and is used extensively in practice. The basic idea is to break encryption into two steps. To encrypt a message m (see Figure 10.1):

1. The sender first chooses a random secret key k, and encrypts k using the public key of the receiver. Call the resulting ciphertext c_1. The receiver will be able to recover k by decrypting c_1, yet k will remain unknown to an eavesdropper (by security of the public-key encryption scheme), and so this has the effect of establishing a shared secret between the sender and the receiver.

2. The sender then encrypts the message m using a *private-key encryption scheme* $(\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ and the secret key k that has just been shared. This results in a ciphertext c_2 that can be decrypted by the receiver using k.

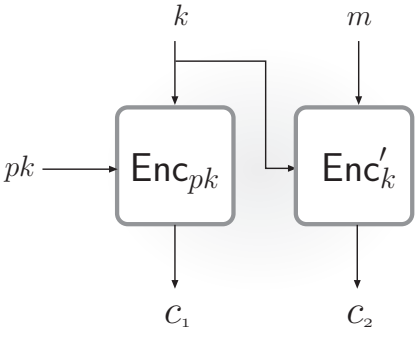

FIGURE 10.1: Hybrid encryption.

The above two steps can be performed in "one shot" by having the sender transmit the ciphertext $\langle c_1, c_2 \rangle$ to the receiver. We stress that although

private-key encryption is used as a component of the construction, the above constitutes a public-key encryption scheme by virtue of the fact that the sender and receiver do not share any secret key *in advance*.

Construction 10.12 gives a formal description of the (public-key) hybrid encryption scheme Π^{hy} based on any public-key encryption scheme Π and private-key encryption scheme Π' (recall that we assume that the key generation algorithm Gen' for Π' just outputs a uniformly distributed key of length n). The construction assumes that Π includes $\{0,1\}^n$ in the underlying plaintext space so that a secret key $k \in \{0,1\}^n$ can be encrypted. The plaintext space for Π^{hy} is identical to the plaintext space of Π', and for simplicity we assume this to be $\{0,1\}^*$.

CONSTRUCTION 10.12

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme, and let $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a private-key encryption scheme. Construct a public-key encryption scheme $\Pi^{\mathsf{hy}} = (\mathsf{Gen}^{\mathsf{hy}}, \mathsf{Enc}^{\mathsf{hy}}, \mathsf{Dec}^{\mathsf{hy}})$ as follows:

- $\mathsf{Gen}^{\mathsf{hy}}$: on input 1^n run $\mathsf{Gen}(1^n)$ and use the public and private keys (pk, sk) that are output.
- $\mathsf{Enc}^{\mathsf{hy}}$: on input a public key pk and a message $m \in \{0,1\}^*$, proceed as follows:
  1. Choose a random $k \leftarrow \{0,1\}^n$ (recall that, by convention, n can be determined from pk).
  2. Compute $c_1 \leftarrow \mathsf{Enc}_{pk}(k)$ and $c_2 \leftarrow \mathsf{Enc}'_k(m)$.
  3. Output the ciphertext $\langle c_1, c_2 \rangle$.
- $\mathsf{Dec}^{\mathsf{hy}}$: on input a private key sk and a ciphertext $\langle c_1, c_2 \rangle$ proceed as follows:
  1. Compute $k := \mathsf{Dec}_{sk}(c_1)$.
  2. Output the message $m := \mathsf{Dec}'_k(c_2)$.

Hybrid encryption.

What is the efficiency of hybrid encryption relative to the approach (as in Equation (10.2)) of using Enc to encrypt bit-by-bit or block-by-block? First observe that hybrid encryption can only possibly give a performance improvement when $|m| > n$; otherwise, instead of encrypting k using Enc we may as well just encrypt m itself. When $|m| \gg n$, though, hybrid encryption gives a substantial efficiency improvement assuming Enc' is more efficient (per bit) that Enc. In detail, for some fixed value of n let α denote the cost of encrypting an n-bit key using Enc, and let β denote the cost (per bit of plaintext) of encryption using Enc'. Then the cost, per bit of plaintext, of encrypting a

t-bit message using Π^{hy} is

$$\frac{\alpha + \beta \cdot t}{t} = \frac{\alpha}{t} + \beta, \tag{10.3}$$

which approaches β as t gets large. In the limit of very long messages, then, the cost per bit incurred by the *public-key* encryption scheme Π^{hy} is the same as the cost per bit incurred by the *private-key* scheme Π'. Hybrid encryption thus allows us to achieve the *functionality* of public-key encryption at the *efficiency* of private-key encryption, at least for sufficiently long messages.

A similar calculation can be used to gauge the effect of hybrid encryption on the ciphertext length. For a fixed value of n, let ℓ denote the length of the encryption of an n-bit key using Enc, and say the private-key encryption of a message m using Enc' results in a ciphertext of length $n + |m|$ (this can be achieved using one of the modes of encryption discussed in Section 3.6.4; actually, even ciphertext length $|m|$ is possible since, as we will see, Π' need not be CPA-secure). Then the total length of a ciphertext in scheme Π^{hy} is

$$\ell + n + |m|. \tag{10.4}$$

When using the approach of Equation (10.2) to encrypt n-bit blocks, the resulting ciphertext has length $\ell \cdot \lceil |m|/n \rceil$, so the above is an improvement for m sufficiently long.

We can use some rough estimates to get a sense for what the above results mean in practice. (We stress that these numbers are only meant to give the reader a feel for the improvement, while actual numbers would depend on a variety of factors.) A typical value for the length n of the (symmetric) key k might be $n \approx 100$. Furthermore, a public-key encryption scheme might encrypt up to 1000 bits in a single invocation, yielding a ciphertext at least twice as long. (More precisely, the ciphertext length in this case for a message of length t would be $2000 \cdot \lceil t/1000 \rceil \approx 2t$ for large t.) Letting α, as before, denote the cost of public-key encryption of a 100-bit key, we see that the approach of Equation (10.2) would encrypt a 1MB ($= 10^6$-bit) message with computational cost $\approx \alpha \cdot 10^6/10^3 = 10^3\alpha$ and the ciphertext would be 2MB long. Compare this to the efficiency of hybrid encryption. Letting β, as before, denote the per-bit computational cost of private-key encryption, a reasonable approximation is $\beta \approx \alpha/10^6$. Using Equation (10.3), we see that the computational cost of hybrid encryption for a 1Mb message would be

$$10^6 \cdot \left(\frac{\alpha}{10^6} + \frac{\alpha}{10^6} \right) = 2\alpha\,,$$

and the ciphertext would be only slightly longer than 1MB. Thus, hybrid encryption improves the computational efficiency in this case by a factor of 500(!), and the ciphertext length by a factor of 2.

It remains to analyze the security of Π^{hy}. In the theorem that follows, we show that the composite scheme Π^{hy} is CPA-secure as long as the original public-key encryption scheme Π is CPA-secure and the private-key scheme Π' has indistinguishable encryptions in the presence of an eavesdropper. Notice that it suffices for Π' to satisfy a weaker definition of security — which, recall, does *not* imply CPA-security in the private-key setting — in order for the hybrid scheme Π^{hy} to be CPA-secure. Intuitively, the reason is that the secret key k used during the course of encryption is chosen *freshly* and completely at random each time a new message is encrypted. Since each key k is used only once, indistinguishability of a single encryption in Π' suffices for security of the hybrid scheme Π^{hy}. (Hybrid encryption is thus one application where a stream cipher may be used, while still achieving strong security guarantees.)

Before formally proving the security of Π^{hy}, we highlight the overall intuition. Let the notation "$X \stackrel{\text{c}}{\equiv} Y$" denote, intuitively, the fact that no polynomial-time adversary can distinguish between X and Y. (This concept is treated more formally in Section 6.8, though we do not rely on that section here.) For example, the fact that Π is CPA-secure means that for any pair of messages m_0, m_1 output by a PPT adversary $\mathcal{A}$ we have

$$(pk, \mathsf{Enc}_{pk}(m_0)) \stackrel{\text{c}}{\equiv} (pk, \mathsf{Enc}_{pk}(m_1)),$$

where pk is generated by $\mathsf{Gen}(1^n)$. That is to say, an encryption of m_0 cannot be distinguished (in polynomial time) from an encryption of m_1, even given pk. Similarly, the fact that Π' has indistinguishable encryptions in the presence of an eavesdropper means that for any m_0, m_1 output by $\mathcal{A}$ we have

$$\mathsf{Enc}'_k(m_0) \stackrel{\text{c}}{\equiv} \mathsf{Enc}'_k(m_1),$$

where k is chosen uniformly at random. Now, in order to prove CPA-security of Π^{hy} we need to show that

$$\big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)\big) \stackrel{\text{c}}{\equiv} \big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)\big) \tag{10.5}$$

for m_0, m_1 output by a PPT adversary $\mathcal{A}$, where pk is output by $\mathsf{Gen}(1^n)$ and the key k is chosen at random from $\{0,1\}^n$. (Equation (10.5) suffices for demonstrating that Π^{hy} has indistinguishable encryptions in the presence of an eavesdropper, and by Theorem 10.10 this implies that Π^{hy} is CPA-secure.)

The proof proceeds in three steps; see Figure 10.2. First we prove that

$$\big(pk,\ \mathsf{Enc}_{pk}(k),\ \mathsf{Enc}'_k(m_0)\big) \stackrel{\text{c}}{\equiv} \big(pk,\ \mathsf{Enc}_{pk}(0^n),\ \mathsf{Enc}'_k(m_0)\big) \tag{10.6}$$

by CPA-security of Π. Indeed, the only difference between the left- and the right-hand sides is the switch from encrypting k to encrypting an all-0 string of the same length, in both cases using scheme Π. But security of Π means that this change is not noticeable by a PPT adversary. Furthermore, this holds even if k is known to the adversary, and so the fact that k is used also to

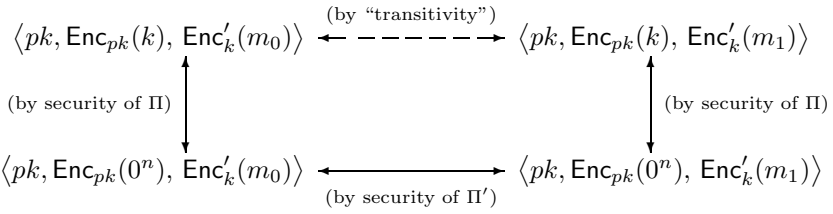

FIGURE 10.2: High-level structure of the proof of Theorem 10.13 (the arrows represent indistinguishability).

encrypt m_0 does not introduce any complications. (In contrast, if we were to try to prove that $\big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)\big) \stackrel{c}{\equiv} \big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)\big)$ based on the security of Π' we would run into trouble, since indistinguishability of $\mathsf{Enc}'_k(m_0)$ and $\mathsf{Enc}'_k(m_1)$ only holds when the adversary has no information about k. However, if k is encrypted using Π then we can no longer claim this. Of course, intuitively $\mathsf{Enc}_{pk}(k)$ does not reveal k, but this is exactly what we are trying to prove.)

Next, we prove that

$$\big(pk, \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)\big) \stackrel{c}{\equiv} \big(pk, \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)\big) \tag{10.7}$$

based on the fact that Π' has indistinguishable encryptions in the presence of an eavesdropper. Indeed, here the difference is between encrypting m_0 and m_1, in both cases using Π' *and a key k chosen uniformly at random*. Furthermore, no other information about k is leaked to the adversary, and in particular no information can be leaked by $\mathsf{Enc}_{pk}(0^n)$ since this is just an encryption of the all-0 string (and not k itself).

Exactly as in the case of Equation (10.6), we can also show that

$$\big(pk, \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)\big) \stackrel{c}{\equiv} \big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)\big)\,, \tag{10.8}$$

by relying again on the CPA-security of Π. Equations (10.6)–(10.8) imply, by transitivity, the desired result in Equation (10.5). (We do not prove that transitivity holds; rather, it will be implicit in the proof we give below.)

THEOREM 10.13 *If Π is a CPA-secure public-key encryption scheme and Π' is a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper, then Π^{hy} as in Construction 10.12 is a CPA-secure public-key encryption scheme.*

PROOF Fix an arbitrary PPT adversary $\mathcal{A}^{\mathsf{hy}}$, and consider experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}^{\mathsf{hy}},\Pi^{\mathsf{hy}}}(n)$. (By Theorem 10.10, once we show that Π^{hy} has indistin-

guishable encryptions in the presence of an eavesdropper we can conclude that it is CPA-secure.) Our goal is to prove that there exists a negligible function negl such that

$$\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}^{\mathsf{hy}},\Pi^{\mathsf{hy}}}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

As previously, we will use the fact that

$$\begin{aligned}\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}^{\mathsf{hy}},\Pi^{\mathsf{hy}}}(n) = 1] = \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0] \\ + \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1].\end{aligned} \tag{10.9}$$

Note that in each case, the probability is taken over randomly-generated pk as well as uniform choice of k. Furthermore, $\mathcal{A}^{\mathsf{hy}}$ is also given the public key pk but this is left implicit for better readability.

Consider the following PPT adversary $\mathcal{A}_1$ that eavesdrops on a message encrypted using public-key scheme Π.

Adversary $\mathcal{A}_1$:

1. $\mathcal{A}_1$, given pk, chooses random $k \leftarrow \{0,1\}^n$ (recall that, by convention, n can be determined from pk) and outputs the pair of messages $k, 0^n$. It is given in return a ciphertext c_1.
2. $\mathcal{A}_1$ runs $\mathcal{A}^{\mathsf{hy}}(pk)$ to obtain two messages m_0, m_1.
3. $\mathcal{A}_1$ computes $c_2 \leftarrow \mathsf{Enc}'_k(m_0)$, then runs $\mathcal{A}^{\mathsf{hy}}(c_1, c_2)$ and outputs the bit b' that is output by $\mathcal{A}^{\mathsf{hy}}$.

When $b = 0$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_1,\Pi}(n)$, the adversary $\mathcal{A}_1$ is given a ciphertext of the form $\mathsf{Enc}_{pk}(k)$ where k was chosen uniformly at random by $\mathcal{A}_1$ in the first step. This means that $\mathcal{A}^{\mathsf{hy}}$ is given a ciphertext of the form $\langle c_1, c_2\rangle = \langle \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)\rangle$, for a randomly-generated pk and uniform choice of k. So,

$$\Pr[\mathcal{A}_1 \text{ outputs } 0 \mid b = 0] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0].$$

On the other hand, when $b = 1$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_1,\Pi}(n)$, adversary $\mathcal{A}_1$ is given a ciphertext of the form $\mathsf{Enc}_{pk}(0^n)$. This means that $\mathcal{A}^{\mathsf{hy}}$ is given a ciphertext of the form $\langle \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'(m_0)\rangle$, and so

$$\Pr[\mathcal{A}_1 \text{ outputs } 1 \mid b = 1] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 1].$$

By the assumption that Π has indistinguishable encryptions in the presence of an eavesdropper, there exists a negligible function negl_1 such that:

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}_1(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_1,\Pi}(n) = 1] \qquad (10.10)\\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}_1 \text{ outputs } 0 \mid b = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}_1 \text{ outputs } 1 \mid b = 1]\\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0]\\ &\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 1].\end{aligned}$$

Next, consider the following PPT adversary $\mathcal{A}'$ that eavesdrops on a message encrypted using the private-key scheme Π'.

Adversary $\mathcal{A}'$:

1. $\mathcal{A}'(1^n)$ runs $\mathsf{Gen}(1^n)$ on its own to generate keys (pk, sk).
2. $\mathcal{A}'$ runs $\mathcal{A}^{\mathsf{hy}}(pk)$ to obtain two messages m_0, m_1. These same messages are output by $\mathcal{A}'$, and it is given in return a ciphertext c_2.
3. $\mathcal{A}'$ computes $c_1 \leftarrow \mathsf{Enc}_{pk}(0^n)$. Then $\mathcal{A}'$ runs $\mathcal{A}^{\mathsf{hy}}(c_1, c_2)$ and outputs the bit b' that is output by $\mathcal{A}^{\mathsf{hy}}$.

When $b = 0$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi'}(n)$, adversary $\mathcal{A}'$ is given a ciphertext of the form $\mathsf{Enc}'_k(m_0)$ where k is chosen at random (and is unknown to $\mathcal{A}'$). This means that $\mathcal{A}^{\mathsf{hy}}$ is given a ciphertext of the form $\langle \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)\rangle$ for a randomly generated pk and a randomly chosen k. We thus have

$$\Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 0].$$

On the other hand, when $b = 1$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi'}(n)$, adversary $\mathcal{A}'$ is given a ciphertext of the form $\mathsf{Enc}'_k(m_1)$ where again k is chosen at random. This means that $\mathcal{A}^{\mathsf{hy}}$ is given a ciphertext of the form $\langle \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)\rangle$ and so

$$\Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 1].$$

By the assumption that Π' has indistinguishable encryptions in the presence of an eavesdropper, there exists a negligible function negl' such that:

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}'(n) &\geq \Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi'}(n) = 1] \qquad (10.11)\\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1]\\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 0]\\ &\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 1].\end{aligned}$$

We will next show something that is exactly analogous to Equation (10.10) (and so the reader is welcome to skip directly to Equation (10.12)). Consider the following PPT adversary $\mathcal{A}_2$ that eavesdrops on a message encrypted using public-key scheme Π.

Adversary $\mathcal{A}_2$:

1. $\mathcal{A}_2$, given pk, chooses random $k \leftarrow \{0,1\}^n$ and outputs the pair of messages $0^n, k$ (the order of these messages has been switched relative to adversary $\mathcal{A}_1$). It is given in return a ciphertext c_1.
2. $\mathcal{A}_2$ runs $\mathcal{A}^{\mathsf{hy}}(pk)$ to obtain two messages m_0, m_1. $\mathcal{A}_2$ computes $c_2 \leftarrow \mathsf{Enc}'_k(m_1)$. Then $\mathcal{A}_2$ runs $\mathcal{A}^{\mathsf{hy}}(c_1, c_2)$ and outputs the bit b' that is output by $\mathcal{A}^{\mathsf{hy}}$.

In experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_2,\Pi}(n)$, when $b = 0$ adversary $\mathcal{A}^{\mathsf{hy}}$ is given a ciphertext of the form $\langle c_1, c_2\rangle = \langle \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)\rangle$ for a randomly generated pk and a uniformly distributed k. Thus,

$$\Pr[\mathcal{A}_2 \text{ outputs } 0 \mid b = 0] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 0].$$

On the other hand, when $b = 1$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_2,\Pi}(n)$, adversary $\mathcal{A}^{\mathsf{hy}}$ is given a ciphertext of the form $\langle \mathsf{Enc}_{pk}(k), \mathsf{Enc}'(m_1)\rangle$, and so

$$\Pr[\mathcal{A}_2 \text{ outputs } 1 \mid b = 1] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1].$$

Since Π has indistinguishable encryptions in the presence of an eavesdropper, there exists a negligible function negl_2 such that:

$$\begin{aligned}
\frac{1}{2} + \mathsf{negl}_2(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_2,\Pi}(n) = 1] \qquad (10.12)\\
&= \frac{1}{2} \cdot \Pr[\mathcal{A}_2 \text{ outputs } 0 \mid b = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}_2 \text{ outputs } 1 \mid b = 1]\\
&= \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 0]\\
&\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1].
\end{aligned}$$

At long last, we come to the conclusion of the proof. Summing Equations (10.10)–(10.12) and using the fact that the sum of three negligible functions is negligible, we see there exists a negligible function negl such that:

$$\begin{aligned}
&\frac{3}{2} + \mathsf{negl}(n) \geq\\
&\frac{1}{2} \cdot \Big(\Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0] + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 1]\\
&\quad + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 0] + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 1]\\
&\quad + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 0] + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1] \Big).
\end{aligned}$$

Note that

$$\Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 1] + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 0] = 1,$$

since the probabilities of complementary events always sum to 1. Similarly,

$$\Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 1] + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 0] = 1.$$

So we see that:

$$\frac{1}{2} + \mathsf{negl}(n) \geq$$
$$\frac{1}{2} \cdot \Big(\Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0] + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1] \Big),$$

which is exactly what we wanted to prove (cf. Equation (10.9)). ■

The above theorem justifies a focus on public-key encryption schemes that can encrypt messages of length n, the security parameter. Of course, in theory it suffices to just construct schemes that encrypt single-bit messages (since they can be extended to encrypt n-bit messages using a bit-by-bit approach as in Section 10.2.2). However, in practice we want more efficient schemes than those yielded by encrypting each bit separately. The point is that even when taking efficiency into account, there is not much reason to worry about encrypting longer messages once a scheme can encrypt messages of length n.

10.4 RSA Encryption

Our discussion regarding public-key encryption has thus far been rather abstract: we have seen how to encrypt arbitrary-length messages using any public-key encryption scheme, but we still have no concrete examples of any such schemes! In this section, we focus on one popular class of schemes based on the *RSA assumption* (cf. Section 7.2.4).

10.4.1 "Textbook RSA" and its Insecurity

Let GenRSA be a PPT algorithm that, on input 1^n, outputs a modulus N that is the product of two n-bit primes, along with integers e, d satisfying $ed = 1 \bmod \phi(N)$. (As usual, the algorithm may fail with negligible probability but we ignore that here.) Recall from Section 7.2.4 that such an algorithm can be easily constructed from any algorithm $\mathsf{GenModulus}$ that outputs a composite modulus N along with its factorization:

ALGORITHM 10.14
RSA key generation GenRSA

Input: Security parameter 1^n
Output: N, e, d as described in the text

$(N, p, q) \leftarrow \mathsf{GenModulus}(1^n)$
$\phi(N) := (p-1)(q-1)$
choose e such that $\gcd(e, \phi(N)) = 1$
compute $d := [e^{-1} \bmod \phi(N)]$
return N, e, d

We present what we call the "textbook RSA" encryption scheme as Construction 10.15. We refer to the scheme as we do since many textbooks describe RSA encryption in exactly this way with no further warning. Unfortunately, "textbook RSA" encryption is *deterministic* and hence automatically insecure as we have already discussed extensively in Section 10.2.1. Even though it is insecure, we show it here since it provides a quick demonstration of why the RSA assumption is so useful for constructing public-key encryption schemes, and it serves as a useful stepping-stone to the secure construction we show in Section 10.4.3. (Presenting the textbook RSA scheme also gives us the opportunity to issue our own warning against using it.) We discuss how the RSA assumption can be used to construct *secure* encryption schemes later in this chapter, and in Chapter 13.

CONSTRUCTION 10.15

Let GenRSA be as in the text. Define a public-key encryption scheme as follows:

- Gen: on input 1^n run $\mathsf{GenRSA}(1^n)$ to obtain N, e, and d. The public key is $\langle N, e \rangle$ and the private key is $\langle N, d \rangle$.
- Enc: on input a public key $pk = \langle N, e \rangle$ and a message $m \in \mathbb{Z}_N^*$, compute the ciphertext

$$c := [m^e \bmod N].$$

- Dec: on input a private key $sk = \langle N, d \rangle$ and a ciphertext $c \in \mathbb{Z}_N^*$, compute the message

$$m := [c^d \bmod N].$$

The "textbook RSA" encryption scheme.

The fact that decryption always succeeds in recovering the message follows immediately from Corollary 7.22. As for the security of the scheme, one thing we can claim is that computing the private key is as hard as factoring

moduli output by GenRSA. The reason for this is that, as mentioned briefly in Section 7.2.4, given $N = pq$ and e, d with $ed = 1 \bmod \phi(N)$, it is possible to compute the factors of N in polynomial time. We emphasize that this result says nothing about whether the message can be recovered from the ciphertext using other means (that do not involve explicit computation of the private key), nor does it imply that the encryption scheme itself is secure.

Although "textbook RSA" is *not* secure with respect to any of the definitions of security we have proposed in this chapter, it is possible to prove a very weak form of security for the scheme if the RSA assumption holds for GenRSA (cf. Definition 7.46). Namely, one can show that if a message m is chosen *uniformly at random* from $\mathbb{Z}_N^*$, then no PPT adversary given the public key $\langle N, e\rangle$ and the resulting ciphertext $c = [m^e \bmod N]$ can recover the entire message m. This is indeed a rather weak guarantee: m must be chosen *at random* (so, in particular, it is not clear what we can say when m corresponds to English-language text), and furthermore the only thing we can claim is that an adversary does not learn *everything* about m (but it may learn a lot of partial information about m). It thus does not constitute a reasonable notion of security for most applications.

RSA Implementation Issues

We close this section with a brief discussion of some practical aspects related to RSA encryption. The discussion here applies not only to the textbook RSA scheme, but also to other schemes that rely on the RSA assumption.

Encoding binary strings as elements of $\mathbb{Z}_N^*$. Let $\ell = \|N\|$. Any binary string m of length $\ell - 1$ can be viewed as an element of $\mathbb{Z}_N$ in the natural way. It is also possible to encode strings of varying lengths as elements of $\mathbb{Z}_N$ by padding using some unambiguous padding scheme.

A theoretical concern is that the (encoded) message m may not lie in $\mathbb{Z}_N^*$ (i.e., it may be the case that $\gcd(m, N) \neq 1$). Even if this occurs, decryption still succeeds as shown in Exercise 7.10. If m is chosen at random, the probability of this event is low (by Proposition B.17). Moreover, even if a sender *tried* to find an m that did not lie in $\mathbb{Z}_N^*$ they would be unable to do so without factoring N: given $m \in \mathbb{Z}_N \setminus \mathbb{Z}_N^*$, the value $\gcd(m, N)$ is a non-trivial factor of N.

Choice of e. There does not appear to be any difference in the hardness of the RSA problem for different exponents e and, as such, different methods have been suggested for selecting e. One popular choice is to set $e = 3$, since then computing eth powers modulo N (as done when encrypting in the textbook RSA scheme) requires only two multiplications (see Appendix B.2.3). If e is to be set equal to 3, then p and q must be chosen to satisfy $p, q \neq 1 \bmod 3$ so that $\gcd(e, \phi(N)) = 1$.

Choosing $e = 3$ leaves the textbook RSA scheme vulnerable to certain attacks, some of which are illustrated in the following section. This should be

taken more as an indication of the inadequacy of Construction 10.15 than as an indication that setting $e = 3$ is a bad choice.

Note that choosing d to be small in order to speed up decryption (that is, changing GenRSA so that a small d is chosen first and then computing e) is a bad idea. If d is chosen in a small range (say, $d < 2^{16}$) then a brute-force search for d is easy to carry out. Even if d is chosen so that $d \approx N^{1/4}$, and so brute-force attacks are ruled out, there are other potential attacks that can be used to recover d from the public key.

Using the Chinese remainder theorem. The receiver, who holds the private key and hence the factorization of N, can utilize the Chinese remainder theorem (Section 7.1.5) to speed up computation of dth roots modulo N, as is necessary when decrypting. Specifically, since we have the correspondence $[c^d \bmod N] \leftrightarrow ([c^d \bmod p], [c^d \bmod q])$, the receiver can compute the partial results

$$m_p := c^d \bmod p = \left[c^{[d \bmod (p-1)]} \bmod p\right]$$

and

$$m_q := c^d \bmod q = \left[c^{[d \bmod (q-1)]} \bmod q\right]$$

and then combine these to obtain $m \leftrightarrow (m_p, m_q)$ as discussed in Section 7.1.5. Note that $[d \bmod (p-1)]$ and $[d \bmod (q-1)]$ can be computed and stored at the time of key generation, and so do not have to be recomputed every time decryption is performed. Assuming that exponentiation modulo a v-bit integer takes v^3 operations, we have that straightforward RSA decryption takes $8n^3$ steps (because $\|N\| = 2n$), whereas using Chinese remaindering it can be carried out in $2n^3$ steps; or 1/4 of the time! Such optimizations that speed up decryption are of great importance because it is often the case that a server needs to carry out many decryption operations simultaneously.

Example 10.16

Say $p = 11$, $q = 23$, and $e = 3$. Then $N = 253$, $\phi(N) = 220$, and $d = 147$.

To encrypt the binary message $m = 0111001$ with textbook RSA and the public key $pk = \langle N = 253, e = 3\rangle$, simply interpret m as the number 57 (and hence an element of $\mathbb{Z}^*_{253}$) in the natural way. Then compute

$$250 := [57^3 \bmod 253].$$

To decrypt, compute $57 := [250^{147} \bmod 253]$. Alternatively, using the Chinese remainder theorem the receiver could compute

$$250^{[147 \bmod 10]} \bmod 11 = 8^7 \bmod 11 = 2$$

and

$$250^{[147 \bmod 22]} \bmod 23 = 20^{15} \bmod 23 = 11.$$

Indeed, $57 \leftrightarrow (2, 11)$ and so decryption succeeds. (The desired answer can be recovered from the representation $(2, 11)$ as described in Section 7.1.5.) $\diamondsuit$

10.4.2 Attacks on Textbook RSA

To obtain more of a feeling for the RSA problem, as well as to illustrate additional problems with textbook RSA encryption, we now describe various attacks, some of which apply only to Construction 10.15 and some of which apply more generally. We emphasize that none of these attacks indicate any vulnerability in provably-secure encryption schemes based on the RSA assumption, such as the one we will see in the next section, when these schemes are used properly.

Encrypting short messages using small e. If e is small then the encryption of "small" messages is insecure when using textbook RSA encryption. For example, say $e = 3$ and the message m is such that $m < N^{1/3}$ but m is otherwise unknown to an attacker. (We assume in this case that m is padded to the left with 0s and then interpreted as an element of $\mathbb{Z}_N$.) In this case, encryption of m does not involve any modular reduction since the integer m^3 is less than N. This means that given the ciphertext $c = [m^3 \bmod N]$ an attacker can determine m by computing $m := c^{1/3}$ *over the integers*, a computation that can be easily carried out.

Note that this is actually a realistic attack scenario: if strings are encoded as elements of $\mathbb{Z}_N^*$, then having $m < N^{1/3}$ corresponds to the encryption of a short message m having length less than $\|N\|/3$ (assuming messages are encoded by padding to the left with 0s). Consider the case of hybrid encryption where 1024-bit RSA is used to encrypt a secret key of length 128 bits; in this case the key may be easily extracted.

A general attack when small e is used.[3] The above attack shows that short messages can be recovered easily from their encryption if textbook RSA with small e is used. Here, we extend the attack to the case of arbitrary-length messages as long as the same message is sent to multiple receivers.

Let $e = 3$ as before, and say the same message m is sent to three different parties holding public keys $pk_1 = \langle N_1, 3\rangle$, $pk_2 = \langle N_2, 3\rangle$, and $pk_3 = \langle N_3, 3\rangle$, respectively. Then an eavesdropper sees

$$c_1 = [m^3 \bmod N_1] \quad \text{and} \quad c_2 = [m^3 \bmod N_2] \quad \text{and} \quad c_3 = [m^3 \bmod N_3].$$

Assume $\gcd(N_i, N_j) = 1$ for all i, j (if not, then at least one of the moduli can be factored immediately and the message m can be easily recovered). Let $N^* = N_1 N_2 N_3$. An extended version of the Chinese remainder theorem says that there exists a unique non-negative value $\hat{c} < N^*$ such that:

$$\begin{aligned} \hat{c} &= c_1 \bmod N_1 \\ \hat{c} &= c_2 \bmod N_2 \\ \hat{c} &= c_3 \bmod N_3. \end{aligned}$$

[3] This attack relies on the Chinese remainder theorem presented in Section 7.1.5.

Moreover, using techniques similar to those shown in Section 7.1.5 it is possible to compute $\hat{c}$ efficiently given the public keys and the above ciphertexts. Note that $\hat{c} = m^3 \bmod N^*$. But since $m < \min\{N_1, N_2, N_3\}$ we have $m^3 < N^*$. As in the previous attack, this means that $\hat{c} = m^3$ *over the integers* (i.e., with no modular reduction taking place), and m can be obtained by computing the integer cube-root of $\hat{c}$.

A quadratic improvement in recovering m. Since textbook RSA encryption is deterministic, we know that if the message m is chosen from a small list of possible values then it is possible to determine m from the ciphertext $c = [m^e \bmod N]$ by simply trying each possible value of m as discussed in Section 10.2.1. If we know that $1 \leq m < \mathcal{L}$ (when interpreting m as an integer), then the time to carry out this attack is linear in $\mathcal{L}$. Thus, one might hope that textbook RSA encryption could be used when $\mathcal{L}$ is large, i.e., the message is chosen from some reasonably-large set of values. One possible scenario where this might occur is in the context of hybrid encryption (see Section 10.3), where the "message" that is encrypted directly by the public-key component is a random secret key of length ℓ, and so $\mathcal{L} = 2^\ell$.

Unfortunately, there is a clever attack that recovers m in this case (with high probability) in time roughly $\sqrt{\mathcal{L}}$. This is a significant improvement in practice: if an 80-bit (random) message is encrypted, then a brute-force attack taking 2^{80} steps is infeasible, but an attack taking 2^{40} steps is relatively easy to carry out.

A description of the attack appears below in Algorithm 10.17. In the description of the algorithm, we assume that $m < 2^\ell$ (i.e., $\mathcal{L} = 2^\ell$) and that the attacker knows ℓ. The value α is a constant with $\frac{1}{2} < \alpha < 1$.

ALGORITHM 10.17
An attack on textbook RSA encryption

Input: Public key $\langle N, e \rangle$; ciphertext c; parameter ℓ
Output: $m < 2^\ell$ such that $m^e = c \bmod N$

set $T := 2^{\alpha\ell}$
for $r = 1$ to T:
  $x_i := [c/r^e \bmod N]$
sort the pairs $\{(r, x_r)\}_{r=1}^{T}$ by their second component
for $s = 1$ to T:
  if $[s^e \bmod N] \stackrel{?}{=} x_r$ for some r
    return $[r \cdot s \bmod N]$

In the algorithm, if an r that is not invertible modulo N is ever encountered, then the factorization of N can be easily computed (because this means that $\gcd(r, N) \neq 1$); we do not bother explicitly writing this in our description of the algorithm.

The time complexity of the algorithm is dominated by the time required to sort the $2^{\alpha\ell}$ pairs (i, x_i); this can be done in time $\mathcal{O}(\ell \cdot 2^{\alpha\ell})$. Binary search is used in the second-to-last line to check whether there exists an r with $x_r = [s^e \bmod N]$.

We now sketch why the attack recovers m with high probability. Let $c = m^e \bmod N$ with $m < 2^\ell$. If m is chosen as a random ℓ-bit integer, it can be shown that with good probability there exist r, s with $1 < r, s \leq 2^{\alpha\ell}$ and $m = r \cdot s$. (See the references at the end of the chapter.) We claim that whenever this occurs, the above algorithm finds m. Indeed, in this case

$$c = m^e = (r \cdot s)^e = r^e \cdot s^e \bmod N,$$

and so $c/r^e = s^e \bmod N$ with $r, s < T$. It is easy to verify that the algorithm finds r, s in this case. If you were not yet convinced, then this attack should tell you never to use deterministic encryption, even if you are encrypting large random keys.

Common modulus attack I. This is a classic example of misuse of RSA. Imagine that a company wants to use the same modulus N for each of its employees. Since it is not desirable for messages encrypted to one employee to be read by any other employee, the company issues different (e_i, d_i) to each employee. That is, the public key of the ith employee is $pk_i = \langle N, e_i \rangle$ and their private key is $sk = \langle N, d_i \rangle$, where $e_i \cdot d_i = 1 \bmod \phi(N)$ for all i.

This approach is insecure, and allows any employee to read messages encrypted to all other employees. The reason is that, as noted in Section 7.2.4, given N and e_i, d_i with $e_i \cdot d_i = 1 \bmod \phi(N)$, the factorization of N can be efficiently computed. Given the factorization of N, of course, it is possible to compute $d_j := e_j^{-1} \bmod \phi(N)$ for any j.

Common modulus attack II. The attack just shown allows any employee to decrypt messages sent to any other employee. This still leaves the possibility that sharing the modulus N is fine as long as all employees trust each other (or, alternatively, as long as confidentiality need only be preserved against outsiders but not against other members of the company). Here we show a scenario indicating that sharing a modulus is still a bad idea, at least when textbook RSA encryption is used.

Say the same message m is encrypted and sent to two different (known) employees with public keys (N, e_1) and (N, e_2) where $e_1 \neq e_2$. Assume further that $\gcd(e_1, e_2) = 1$. Then an eavesdropper sees the two ciphertexts

$$c_1 = m^{e_1} \bmod N \quad \text{and} \quad c_2 = m^{e_2} \bmod N.$$

Since $\gcd(e_1, e_2) = 1$, there exist integers X, Y such that $Xe_1 + Ye_2 = 1$ by Proposition 7.2. Moreover, given the public exponents e_1 and e_2 it is possible to efficiently compute X and Y using the extended Euclidean algorithm (see Appendix B.1.2). We claim that $m = [c_1^X \cdot c_2^Y \bmod N]$, which can easily be

calculated. This is true because

$$c_1^X \cdot c_2^Y = m^{Xe_1} m^{Ye_2} = m^{Xe_1 + Ye_2} = m^1 = m \bmod N.$$

Thus it is much better to share the complete key than part of it.

This example and those preceding it should serve as a warning to only ever use RSA (and any other cryptographic scheme) in the exact way that it is specified. Even minor and seemingly harmless modifications can open the door to attack.

10.4.3 Padded RSA

The insecurity of the textbook RSA encryption scheme, both vis-a-vis the various attacks described in the previous two sections as well as the fact that it cannot possibly satisfy Definition 10.3, means that other approaches to encryption based on RSA must be considered. One simple idea is to *randomly pad* the message before encrypting. A general paradigm for this approach is shown as Construction 10.18. The construction is defined based on a parameter ℓ that determines the length of messages that can be encrypted.

CONSTRUCTION 10.18

Let GenRSA be as before, and let ℓ be a function with $\ell(n) \leq 2n - 2$ for all n. Define a public-key encryption scheme as follows:

- Gen: on input 1^n, run $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d). Output the public key $pk = \langle N, e \rangle$, and the private key $sk = \langle N, d \rangle$.
- Enc: on input a public key $pk = \langle N, e \rangle$ and a message $m \in \{0,1\}^{\ell(n)}$, choose a random string $r \leftarrow \{0,1\}^{\|N\| - \ell(n) - 1}$ and interpret $r \| m$ as an element of $\mathbb{Z}_N$ in the natural way. Output the ciphertext
$$c := [\,(r \| m)^e \bmod N].$$
- Dec: on input a private key $sk = \langle N, d \rangle$ and a ciphertext $c \in \mathbb{Z}_N^*$, compute
$$\hat{m} := [c^d \bmod N],$$
and output the $\ell(n)$ low-order bits of $\hat{m}$.

The padded RSA encryption scheme.

It is clear from the description of the scheme that decryption always succeeds. (This is immediate when $r \| m \in \mathbb{Z}_N^*$, but is true even if $r \| m \notin \mathbb{Z}_N^*$. In any case, the latter occurs with only negligible probability.) Security of the padded RSA encryption scheme depends on ℓ. If ℓ is too large, so that $\ell(n) = 2n - \mathcal{O}(\log n)$, then a brute-force search through all possible values of the random padding r can be carried out in $2^{\mathcal{O}(\log n)} = \mathsf{poly}(n)$ time and the scheme will not be CPA-secure. This is not "just" a theoretical concern, since

it also implies that if the message m is chosen from a small space of possibilities then an eavesdropper can use a brute-force search to determine m.

When $\ell(n) = c \cdot n$ for some constant $c < 2$, it is reasonable to conjecture that padded RSA is secure but there is no known proof of security based on the standard RSA assumption introduced in Chapter 7. (It is possible, however, to prove security in this case based on a non-standard assumption; see Exercise 10.9.)

When $\ell(n)$ is very small, it is possible to prove the following:

THEOREM 10.19 *If the RSA problem is hard relative to* GenRSA *then Construction 10.18 with $\ell(n) = \mathcal{O}(\log n)$ has indistinguishable encryptions under a chosen-plaintext attack.*

A full proof of this theorem is beyond the scope of this book; however, one step of the proof for $\ell(n) = 1$ is given as part of Exercise 10.10.[4] For a different, but somewhat related, way to construct a secure public-key encryption scheme based on the hardness of RSA, see Section 10.7.

PKCS #1 v1.5. A widely-used and standardized encryption scheme, *RSA Laboratories Public-Key Cryptography Standard (PKCS) #1 version 1.5*, utilizes what is essentially padded RSA encryption. For a public key $pk = \langle N, e \rangle$ of the usual form, let k denote the length of N in bytes; i.e., k is the integer satisfying $2^{8(k-1)} \leq N < 2^{8k}$. Messages m to be encrypted are assumed to be a multiple of 8 bits long, and can have length up to $k - 11$ bytes. Encryption of a message m that is D-bytes long is computed as

$$[(00000000 \| 00000010 \| r \| 00000000 \| m)^e \bmod N],$$

where r is a randomly-generated string of $(k - D - 3)$ bytes, with none of these bytes equal to 0. (This latter condition on r simply enables the message to be unambiguously recovered upon decryption.) Note that the maximum allowed length of m ensures that the length of r is at least 8 bytes.

PKCS #1 v1.5 is believed to be CPA-secure, although no proof based on the RSA assumption has ever been shown. Subsequent to the introduction of PKCS #1 v1.5, a chosen-ciphertext attack on this scheme was demonstrated. This motivated a change in the standard to a newer scheme called OAEP (for Optimal Asymmetric Encryption Padding) that has been proven secure against such attacks (in what is called the random oracle model; see Chapter 13). This updated version is preferred for new implementations, though the older version is still widely used for reasons of backwards-compatibility. We present a high-level description of this scheme in Section 13.2.3.

[4] For those who have covered Chapter 6, we remark that the theorem relies on the fact that the least-significant bit is a hard-core predicate for RSA.

10.5 The El Gamal Encryption Scheme

The El Gamal encryption scheme is another popular encryption scheme, and its security can be based on the hardness of the decisional Diffie-Hellman (DDH) problem. The DDH problem is discussed in detail in Section 7.3.2.

We begin by stating and proving a simple lemma that underlies the El Gamal encryption scheme and will also be useful for our work in Chapter 11. Let $\mathbb{G}$ be a finite group, and let $m \in \mathbb{G}$ be an arbitrary element. Essentially, the lemma states that multiplying m by a random group element g yields a random group element g'. Since the distribution of g' is independent of m, this means that g' *contains no information about* m. That is:

LEMMA 10.20 *Let $\mathbb{G}$ be a finite group, and let $m \in \mathbb{G}$ be an arbitrary element. Then choosing random $g \leftarrow \mathbb{G}$ and setting $g' := m \cdot g$ gives the same distribution for g' as choosing random $g' \leftarrow \mathbb{G}$. I.e., for any $\hat{g} \in \mathbb{G}$*

$$\Pr[m \cdot g = \hat{g}] = 1/|\mathbb{G}|,$$

where the probability is taken over random choice of g.

PROOF Let $\hat{g} \in \mathbb{G}$ be arbitrary. Then

$$\Pr[m \cdot g = \hat{g}] = \Pr[g = m^{-1} \cdot \hat{g}].$$

Since g is chosen uniformly at random, the probability that g is equal to the fixed element $m^{-1} \cdot \hat{g}$ is exactly $1/|\mathbb{G}|$. ■

The above lemma suggests a way to construct a perfectly-secret *private-key* encryption scheme that encrypts messages in $\mathbb{G}$. The sender and receiver share as their secret key a random element $g \leftarrow \mathbb{G}$. Then, to encrypt the message $m \in \mathbb{G}$, the sender computes the ciphertext $g' := m \cdot g$. The receiver can recover the message from the ciphertext g' by computing $m := g'/g$. Perfect secrecy of this schemes follows immediately from the lemma. In fact, we have already seen this scheme in a different guise — the one-time pad encryption scheme is exactly an example of the above approach, with the underlying group being the set of strings of some fixed length under the group operation of bit-wise XOR.

In the scheme as we have described it, g is truly random and decryption is possible only because the sender and receiver have shared g in advance. In the public-key setting, a different technique is needed to allow the receiver to decrypt. The crucial idea is to use a "pseudorandom" element g rather than a truly random one. In a bit more detail, g will be defined in such a way that the receiver will be able to compute g using her private key, yet g will "look

random" for any eavesdropper. We now see how to implement this idea using the DDH assumption, and this discussion should become even more clear once we see the proof of Theorem 10.22.

As in Section 7.3.2, let $\mathcal{G}$ be a polynomial-time algorithm that, on input 1^n, outputs a description of a cyclic group $\mathbb{G}$, its order q (with $\|q\| = n$), and a generator g. (As usual, we also require that the group operation in $\mathbb{G}$ can be computed in time polynomial in n, and we allow that $\mathcal{G}$ may fail with negligible probability.) One use of El Gamal encryption is with $\mathbb{G}$ being an elliptic curve group; such groups were introduced briefly in Section 7.3.4 and have the advantage of enabling the use of much shorter keys.

The El Gamal encryption scheme is defined as follows:

CONSTRUCTION 10.21

Let $\mathcal{G}$ be as in the text. Define a public-key encryption scheme as follows:

- Gen: on input 1^n run $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$. Then choose a random $x \leftarrow \mathbb{Z}_q$ and compute $h := g^x$. The public key is $\langle \mathbb{G}, q, g, h \rangle$ and the private key is $\langle \mathbb{G}, q, g, x \rangle$.
- Enc: on input a public key $pk = \langle \mathbb{G}, q, g, h \rangle$ and a message $m \in \mathbb{G}$, choose a random $y \leftarrow \mathbb{Z}_q$ and output the ciphertext

$$\langle g^y,\ h^y \cdot m \rangle.$$

- Dec: on input a private key $sk = \langle \mathbb{G}, q, g, x \rangle$ and a ciphertext $\langle c_1, c_2 \rangle$, output

$$m := c_2 / c_1^x.$$

The El Gamal encryption scheme.

To see that decryption succeeds, let $\langle c_1, c_2 \rangle = \langle g^y, h^y \cdot m \rangle$ with $h = g^x$. Then

$$\frac{c_2}{c_1^x} = \frac{h^y \cdot m}{(g^y)^x} = \frac{(g^x)^y \cdot m}{g^{xy}} = \frac{g^{xy} \cdot m}{g^{xy}} = m.$$

To fully specify the scheme, we need to show how to encode binary strings as elements of $\mathbb{G}$. We discuss this at the end of this section.

We now prove the security of the El Gamal encryption scheme. The reader may want to compare the proof of the following theorem to the proofs of Theorems 3.16 and 9.3.

THEOREM 10.22 *If the DDH problem is hard relative to $\mathcal{G}$, then the El Gamal encryption scheme has indistinguishable encryptions under a chosen-plaintext attack.*

PROOF Let Π denote the El Gamal encryption scheme. We prove that Π has indistinguishable encryptions in the presence of an eavesdropper. By Theorem 10.10 this implies that it is CPA-secure.

Let $\mathcal{A}$ be a probabilistic, polynomial-time adversary, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1].$$

Consider the modified "encryption scheme" $\widetilde{\Pi}$ where Gen is the same as in Π, but encryption of a message m with respect to the public key $\langle \mathbb{G}, q, g, h \rangle$ is done by choosing random $y \leftarrow \mathbb{Z}_q$ and $z \leftarrow \mathbb{Z}_q$ and outputting the ciphertext

$$\langle g^y,\ g^z \cdot m \rangle.$$

Although $\widetilde{\Pi}$ is not actually an encryption scheme (as there is no way for the receiver to decrypt), the experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$ is still well-defined since that experiment depends only on the key generation and encryption algorithms.

Lemma 10.20 and the discussion that immediately follows it imply that the second component of the ciphertext in scheme $\widetilde{\Pi}$ is a uniformly-distributed group element and, in particular, is independent of the message m being encrypted. (Remember that g^z is a random element of $\mathbb{G}$ when z is chosen at random from $\mathbb{Z}_q$.) The first component of the ciphertext is trivially independent of m. Taken together, this means that the entire ciphertext contains no information about m. It follows that

$$\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] = \frac{1}{2}.$$

Now consider the following PPT algorithm D that attempts to solve the DDH problem relative to $\mathcal{G}$ (recall that D receives $(\mathbb{G}, q, g, g_1, g_2, g_3)$ where $g_1 = g^x$, $g_2 = g^y$, and g_3 equals either g^{xy} or g^z, for random x, y, z):

> **Algorithm** D:
> The algorithm is given $\mathbb{G}, q, g, g_1, g_2, g_3$ as input.
>
> - Set $pk = \langle \mathbb{G}, q, g, g_1 \rangle$ and run $\mathcal{A}(pk)$ to obtain two messages m_0, m_1.
> - Choose a random bit b, and set $c_1 := g_2$ and $c_2 := g_3 \cdot m_b$.
> - Give the ciphertext $\langle c_1, c_2 \rangle$ to $\mathcal{A}$ and obtain an output bit b'. If $b' = b$, output 1; otherwise, output 0.

Let us analyze the behavior of D. There are two cases to consider:

Case 1: Say the input to D is generated by running $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$, then choosing random $x, y, z \leftarrow \mathbb{Z}_q$, and finally setting $g_1 := g^x$, $g_2 := g^y$, and $g_3 := g^z$. Then D runs $\mathcal{A}$ on a public key constructed as

$$pk = \langle \mathbb{G}, q, g, g^x \rangle$$

and a ciphertext constructed as

$$\langle c_1, c_2 \rangle = \langle g^y,\ g^z \cdot m_b \rangle.$$

We see that in this case the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to $\mathcal{A}$'s view in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n)$. Since D outputs 1 exactly when the output b' of $\mathcal{A}$ is equal to b, we have that

$$\Pr[D(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] = \frac{1}{2}.$$

Case 2: Say the input to D is generated by running $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$, then choosing random $x, y \leftarrow \mathbb{Z}_q$, and finally setting $g_1 := g^x$, $g_2 := g^y$, and $g_3 := g^{xy}$. Then D runs $\mathcal{A}$ on a public key constructed as

$$pk = \langle \mathbb{G}, q, g, g^x \rangle$$

and a ciphertext constructed as

$$\langle c_1, c_2 \rangle = \langle g^y,\ g^{xy} \cdot m_b \rangle = \langle g^y,\ (g^x)^y \cdot m_b \rangle.$$

We see that in this case the view of $\mathcal{A}$ when run as a subroutine by D is distributed exactly as $\mathcal{A}$'s view in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. Since D outputs 1 exactly when the output b' of $\mathcal{A}$ is equal to b, we have that

$$\Pr[D(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n).$$

Since the DDH problem is hard relative to $\mathcal{G}$, there must exist a negligible function negl such that

$$\begin{aligned}\mathsf{negl}(n) &\geq \left| \Pr[D(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] - \Pr[D(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1] \right| \\ &= \left| \frac{1}{2} - \varepsilon(n) \right|.\end{aligned}$$

This implies that $\varepsilon(n) \leq \frac{1}{2} + \mathsf{negl}(n)$, completing the proof. ■

El Gamal Implementation Issues

We briefly discuss a few practical aspects related to El Gamal encryption.

Encoding binary strings. As noted earlier, in order to fully specify a usable encryption scheme we need to show how to encode binary strings as elements of $\mathbb{G}$. This, of course, depends on the particular type of group under consideration. We sketch one possible encoding when $\mathbb{G}$ is taken to be the subgroup of quadratic residues modulo a strong prime p as discussed in Section 7.3.3. The encoding we present was chosen for simplicity, and more efficient encodings are possible.

Let p be a strong prime, i.e., $q = (p-1)/2$ is also prime. Then the set of quadratic residues modulo p forms a group $\mathbb{G}$ of order q under multiplication modulo p. We can map the integers $\{1, \ldots, (p-1)/2\}$ to the set of quadratic residues modulo p by squaring: that is, the integer $\tilde{m}$ is mapped to the quadratic residue $m = [\tilde{m}^2 \bmod p]$. This encoding is one-to-one and efficiently reversible. It is one-to-one since any quadratic residue $[\tilde{m}^2 \bmod p]$ has exactly the two square roots $[\pm\tilde{m} \bmod p]$, and exactly one of these values lies in the range $\{1, \ldots, (p-1)/2\}$. (This is so because $[\tilde{m} \bmod p] \leq (p-1)/2$ if and only if $[-\tilde{m} \bmod p] = p - [\tilde{m} \bmod p] > (p-1)/2$.) The encoding is efficiently invertible since square roots modulo p are easy to compute (the interested reader is referred to Sections 11.1.1 and 11.2.1).

Given the above, we can map a string $\hat{m}$ of length $n-1$ to an element $m \in \mathbb{G}$ in the following way (recall that $n = \|q\|$): given a string $\hat{m} \in \{0,1\}^{n-1}$, interpret it as an integer in the natural way and add 1 to obtain an integer $\tilde{m}$ with $1 \leq \tilde{m} \leq q$ (recall $q = (p-1)/2$). Then take $m = [\tilde{m}^2 \bmod p]$.

Example 10.23

Let $q = 83$ and $p = 2q+1 = 167$, and let $\mathbb{G}$ be the group of quadratic residues modulo p. Since q is prime, any element of $\mathbb{G}$ except 1 is a generator; take $g = 2^2 = 4 \bmod 167$. Say the receiver chooses secret key $37 \in \mathbb{Z}_{83}$ and so the public key is

$$pk = \langle 167, 83, 4, [4^{37} \bmod 167] \rangle = \langle 167, 83, 4, 76 \rangle.$$

To encrypt the 6-bit message $\hat{m} = 011101$, view it as the integer 29 and then add 1 to obtain $\tilde{m} = 30$. Squaring this gives $m = [30^2 \bmod 167] = 65$. This is our encoding of the message. Picking $y = 71$ when encrypting, we obtain the ciphertext

$$\langle [4^{71} \bmod 167],\ [76^{71} \cdot 65 \bmod 167] \rangle = \langle 132,\ 44 \rangle.$$

To decrypt, the receiver first computes $124 = [132^{37} \bmod 167]$; then, since $66 = [124^{-1} \bmod 167]$, the receiver can recover $m = 65 = [44 \cdot 66 \bmod 167]$. This m has the two square roots 30 and 137, but the latter is greater than q. So $\tilde{m} = 30$ and $\hat{m}$ is determined to be the binary representation of 29. $\diamondsuit$

Sharing public parameters. Our description of the El Gamal encryption scheme in Construction 10.21 requires the receiver to run $\mathcal{G}$ to generate $\mathbb{G}, q, g$. In practice, however, it is common for these parameters to be generated "once-and-for-all" and then used by multiple receivers. (For example, a system administrator can fix these parameters for a particular choice of security parameter n, and then everyone in the system can share these values.) Of course, each receiver must choose their own secret value x and publish their own public key containing $h = g^x$.

Sharing public parameters is not believed to compromise the security of the encryption scheme in any way. Assuming that the DDH problem is hard relative to $\mathcal{G}$ in the first place, there is no problem using shared public parameters because the DDH problem is still believed to be hard even for the party who runs $\mathcal{G}$ to generate $\mathbb{G}, q, g$. This is in contrast to RSA, where the party who runs GenRSA, at least the way we have described it, knows the factorization of the modulus N that is output. Other attacks when parameters are shared were described in Section 10.4.2. We conclude that in the case of RSA, parameters *cannot* be shared.

10.6 Security Against Chosen-Ciphertext Attacks

Chosen-ciphertext attacks, in which an adversary is allowed to obtain the decryption of arbitrary ciphertexts of its choice (with one technical restriction described below), are as much of a concern in the public-key setting as they are in the private-key setting. Arguably, in fact, they are *more* of a concern in the public-key setting since a receiver in the public-key setting expects to receive ciphertexts from multiple senders, who are possibly unknown in advance, whereas a receiver in the private-key setting intends only to communicate with a single, known sender using any particular secret key.

There are a number of realistic scenarios in which chosen-ciphertext attacks are possible. Assume an eavesdropper $\mathcal{A}$ observes a ciphertext c sent by a sender $\mathcal{S}$ to a receiver $\mathcal{R}$. In the public-key setting one can imagine two broad classes of chosen-ciphertext attacks that might occur:

- $\mathcal{A}$ might send a ciphertext c' to $\mathcal{R}$, but *claim that this ciphertext was sent by* $\mathcal{S}$. (E.g., in the context of encrypted e-mail, $\mathcal{A}$ might construct an encrypted e-mail message c' and forge the "`From`" field so that it appears that the e-mail originated from $\mathcal{S}$.) In this case, although it is unlikely that $\mathcal{A}$ would be able to obtain the entire decryption m' of c', it might be possible for $\mathcal{A}$ to infer some information about m' based on the subsequent behavior of $\mathcal{R}$. By generating c' from c, it may be possible to infer information about the message m from the decryption of c' to m'.

- $\mathcal{A}$ might send a ciphertext c' to $\mathcal{R}$ *in its own name*. In this case, it may be easier for $\mathcal{A}$ to obtain the entire decryption m' of c' because $\mathcal{R}$ may respond directly to $\mathcal{A}$. Another possibility is that $\mathcal{A}$ may not obtain the decryption of c' at all, but the content of m' may have beneficial consequences for $\mathcal{A}$ (see the third scenario below for an example).

Note that the second class of attacks applies only in the context of public-key encryption, and does not really make sense in the private-key setting. We now

give some scenarios demonstrating the above types of attacks. In each case, we also discuss the effect of replaying c itself (i.e., setting $c' = c$) to illustrate why the definitional restriction of disallowing such a query still results in a practically-meaningful security notion.

Scenario 1. Say a user $\mathcal{S}$ logs in to her bank account by sending an encryption of her password pw to the bank. (Logging in this way has other problems, but we ignore these for now.) Assume further that there are two types of error messages that the bank sends after a failed login: upon receiving an encryption of pw from a user $\mathcal{S}$, who is assumed to have an account with the bank, the bank sends "invalid password" if pw contains any non-alphanumeric characters, and returns "password incorrect" if pw is a valid password but it does not match the stored password of $\mathcal{S}$.

If an adversary obtains a ciphertext c sent by $\mathcal{S}$ to the bank, the adversary can now mount a (partial) chosen-ciphertext attack by sending ciphertexts c' to the bank on behalf of $\mathcal{S}$, and observing the error messages that result. This information may be enough to enable the adversary to determine the user's password. Note that the adversary gains no information about the user's password by sending c to the bank, since it already knows in this case that no error message will be generated.[5]

Scenario 2. Say $\mathcal{S}$ sends an encrypted e-mail c to $\mathcal{R}$, and this e-mail is observed by $\mathcal{A}$. If $\mathcal{A}$ sends (in its own name) an encrypted e-mail c' to $\mathcal{R}$, then $\mathcal{R}$ might reply to this e-mail *and quote the decrypted text m' corresponding to c'*. In this case, $\mathcal{R}$ is exactly acting as a decryption oracle for $\mathcal{A}$ and might potentially decrypt any ciphertext that $\mathcal{A}$ sends it. On the other hand, if $\mathcal{A}$ sends c itself to $\mathcal{R}$ then $\mathcal{R}$ may get suspicious and refuse to respond, depending on the contents of the underlying message m.

Scenario 3. A closely related issue to that of chosen-ciphertext security is the possible *malleability* of ciphertexts. Since a formal definition is quite involved, we do not pursue one here but instead only give the intuitive idea. Say an encryption scheme has the property that given an encryption c of some unknown message m, it is possible to come up with a ciphertext c' that is an encryption of a message m' *that is unknown, but related in some known way to m*. For example, perhaps given an encryption c of m, it is possible to construct a ciphertext c' that is an encryption of $2m$. (We will see natural examples of schemes with this and similar properties later in this section. See also Section 11.3.3.)

Now imagine that $\mathcal{R}$ is running an auction, where two parties $\mathcal{S}$ and $\mathcal{A}$ submit their bids by encrypting them using the public key of $\mathcal{R}$. If a CPA-secure

[5] Re-sending c is an example of a *replay attack*, and in this example would have the negative effect of fooling the bank into thinking that $\mathcal{S}$ was logging in. This can be fixed by having the user encrypt a time-stamp along with her password, but the addition of a time-stamp does not change the feasibility of the described attack. (In fact, it may make an attack easier since there may now be a third error message indicating an invalid time-stamp.)

encryption scheme having the above property is used, it may be possible for an adversary $\mathcal{A}$ to always place the highest bid (without bidding the maximum) by carrying out the following attack: wait until $\mathcal{S}$ sends a ciphertext c corresponding to its bid m (that is unknown to $\mathcal{A}$); then, send a ciphertext c' corresponding to the bid $m' = 2m$. Note that both m and m' remain unknown to $\mathcal{A}$ (until $\mathcal{R}$ announces the results), and so the possibility of such an attack does not contradict the fact that the encryption scheme is CPA-secure. In contrast, it can be shown that CCA-secure schemes are *non-malleable* meaning that they are not vulnerable to such attacks.

The definition. We now present a formal definition of security against chosen-ciphertext attacks that exactly parallels Definition 3.30. (A cosmetic difference is that we do not give the adversary access to an encryption oracle; as discussed previously, an encryption oracle does not give any additional power to the adversary in the public-key setting.) For a public-key encryption scheme Π and an adversary $\mathcal{A}$, consider the following experiment:

The CCA indistinguishability experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$**:**

1. $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
2. *The adversary* $\mathcal{A}$ *is given* pk *and access to a decryption oracle* $\mathsf{Dec}_{sk}(\cdot)$. *It outputs a pair of messages* m_0, m_1 *of the same length. (These messages must be in the plaintext space associated with* pk.)
3. *A random bit* $b \leftarrow \{0,1\}$ *is chosen, and then a ciphertext* $c \leftarrow \mathsf{Enc}_{pk}(m_b)$ *is computed and given to* $\mathcal{A}$.
4. $\mathcal{A}$ *continues to interact with the decryption oracle, but may not request a decryption of* c *itself. Finally,* $\mathcal{A}$ *outputs a bit* b'.
5. *The output of the experiment is defined to be 1 if* $b' = b$, *and 0 otherwise.*

DEFINITION 10.24 *A public-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable encryptions under a chosen-ciphertext attack (*or is* CCA-secure) *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n).$$

We will not show any examples of CCA-secure encryption schemes in this book, since proofs of security for all existing constructions are rather complex. We remark, however, that a CCA-secure encryption scheme based on the DDH assumption and with efficiency roughly twice that of El Gamal encryption is known (see the references at the end of this chapter). In Chapter 13 we discuss CCA-secure encryption schemes that are efficient, but that can be proven

secure only in an idealized model explained in detail there. Constructing simpler or more efficient CCA-secure public-key encryption schemes, especially based on the RSA assumption, is an important open problem.

Examples of Chosen-Ciphertext Attacks

The vulnerability of encryption schemes to chosen-ciphertext attacks is not only a theoretical possibility. We show here that all the schemes we have seen so far are insecure under such attacks.

Textbook RSA encryption. In our earlier discussion, we noted that the textbook RSA encryption scheme at least satisfies the following security property (assuming the RSA problem is hard for GenRSA): if a message m is chosen uniformly at random from $\mathbb{Z}_N^*$ and encrypted with respect to the public key $\langle N, e\rangle$, then an eavesdropping adversary cannot recover m in its entirety. It is not hard to see that even this weak property no longer holds if the adversary is allowed to mount a chosen-ciphertext attack. Say an adversary $\mathcal{A}$ intercepts the ciphertext $c = [m^e \bmod N]$. Then the adversary can choose a random $r \leftarrow \mathbb{Z}_N^*$ and compute the ciphertext $c' = [r^e \cdot c \bmod N]$. Given the decryption m' of this ciphertext, $\mathcal{A}$ can recover $m = [m' \cdot r^{-1} \bmod N]$. To see that this works, note that

$$m' \cdot r^{-1} = (c')^d r^{-1} = (r^e \cdot m^e)^d r^{-1} = r^{ed} m^{ed} r^{-1} = r m r^{-1} = m \bmod N,$$

where d is the value contained in the receiver's private key and used to decrypt c' (and so $ed = 1 \bmod \phi(N)$).

Textbook RSA encryption is also vulnerable to the exact attack shown in the auction scenario discussed earlier (Scenario 3). Say an adversary observes a ciphertext $c = [m^e \bmod N]$ encrypted with respect to the public key $\langle N, e\rangle$. Then we claim that the ciphertext $c' = [2^e c \bmod N]$ decrypts to $[2m \bmod N]$. This holds because

$$(c')^d = (2^e m^e)^d = 2^{ed} m^{ed} = 2m \bmod N,$$

where d is as above.

PKCS #1 v1.5. Recall that the public-key encryption scheme used as part of the PKCS #1 v1.5 standard uses a variant of padded RSA encryption where a portion of the padding is done in a specific way (and cannot consist of arbitrary bits). If a ciphertext is decrypted and discovered not to have the correct format, an error message is returned. It turns out that the presence of these error messages is sufficient to enable a chosen-ciphertext attack against the scheme. That is, given a properly-generated ciphertext c, an attacker can recover the underlying message m by submitting multiple ciphertexts c' and observing *only* which ciphertexts are decrypted successfully and which generate an error. Since this sort of information is easy to obtain (for example, from servers that issue error messages when they receive incorrectly formatted

messages), the attack is practical (though the number of ciphertexts needed for the attack is large).

The existence of such a practical chosen-ciphertext attack on the scheme came as somewhat of a surprise, and prompted efforts to standardize an improved encryption scheme that could be proven secure against chosen-ciphertext attacks. These efforts culminated in (a variant of) a scheme that we discuss in Section 13.2.3.

El Gamal encryption. The El Gamal encryption scheme is as vulnerable to chosen-ciphertext attacks as textbook RSA encryption is. This may be somewhat surprising since we have proved that the El Gamal encryption scheme is CPA-secure under the DDH assumption, but we emphasize again that there is no contradiction since we are now considering a stronger attack model.

Say an adversary $\mathcal{A}$ intercepts a ciphertext $c = \langle c_1, c_2 \rangle$ that is an encryption of the (encoded) message m with respect to the public key $pk = \langle \mathbb{G}, q, g, h \rangle$. This means that

$$c_1 = g^y \quad \text{and} \quad c_2 = h^y \cdot m$$

for some $y \in \mathbb{Z}_q$ unknown to $\mathcal{A}$. Nevertheless, if the adversary computes $c_2' := c_2 \cdot m'$ then it is easy to see that the ciphertext $c' = \langle c_1, c_2' \rangle$ is an encryption of the message $m \cdot m'$. This observation leads to an easy chosen-ciphertext attack, and also shows that El Gamal encryption is vulnerable in the auction scenario mentioned above.

One might object that the receiver will become suspicious if it receives two ciphertexts c, c' that share the same first component. (Indeed, for honestly-generated ciphertexts this occurs with negligible probability.) However, this is easy for the adversary to avoid. Letting c_1, c_2, m, m' be as above, $\mathcal{A}$ can choose a random $y'' \leftarrow \mathbb{Z}_q$ and set $c_1'' := c_1 \cdot g^{y''}$ and $c_2'' := c_2 \cdot h^{y''} \cdot m'$. Then

$$c_1'' = g^y \cdot g^{y''} = g^{y+y''} \quad \text{and} \quad c_2'' = h^y m \cdot h^{y''} m' = h^{y+y''} mm',$$

and so the ciphertext $c'' = \langle c_1'', c_2'' \rangle$ is again an encryption of $m \cdot m'$ but with a completely random first component.

10.7 * Trapdoor Permutations

(This section relies on the material presented in Section 7.4.1.)

In Section 10.4.3 we saw how to construct a CPA-secure public-key encryption scheme based on the RSA assumption. By distilling those properties of RSA that are used in the construction, and defining an abstract notion that encapsulates those properties, we can hope to obtain a general *template* for constructing secure encryption schemes based on any primitive satisfying the

same set of properties. *Trapdoor permutations*, which are a special case of one-way permutations, serve as one such abstraction.

In the following section, we define families of trapdoor permutations and observe that the RSA family of one-way permutations (Construction 7.71) satisfies the additional requirements needed to be a family of *trapdoor* permutations. In Section 10.7.2 we show how a public-key encryption scheme can be constructed from any trapdoor permutation. Apart from in Section 10.7.2, the material in Section 10.7.1 is used directly only in Section 11.2, where a second example of a trapdoor permutation is shown; trapdoor permutations are mentioned in passing in Chapter 13 but are not essential for understanding the material there. Section 10.7.2 is not used in the rest of the book.

10.7.1 Definition

Recall the definitions of families of functions and families of one-way permutations from Section 7.4.1. In that section, we showed that the RSA assumption naturally gives rise to a family of one-way permutations. The astute reader may have noticed that the construction we gave (Construction 7.71) has a special property that was not remarked upon there: namely, the parameter generation algorithm Gen outputs some additional information along with I that *enables efficient inversion of* f_I. We refer to such additional information as a *trapdoor*, and call families of one-way permutations with this additional property *families of trapdoor permutations*. A formal definition follows.

DEFINITION 10.25 *A tuple of polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Samp}, f, \mathsf{Inv})$ *is a* family of trapdoor permutations *(or a* trapdoor permutation*) if the following hold:*

- *The probabilistic* parameter generation algorithm Gen*, on input* 1^n*, outputs* (I, td) *with* $|I| \geq n$*. Each* (I, td) *output by* Gen *defines a set* $\mathcal{D}_I = \mathcal{D}_{\mathsf{td}}$.
- *Let* Gen_1 *denote the algorithm that results by running* Gen *and outputting only* I*. Then* $(\mathsf{Gen}_1, \mathsf{Samp}, f)$ *is a family of one-way permutations.*
- *The deterministic* inverting algorithm Inv*, on input* td *and* $y \in \mathcal{D}_{\mathsf{td}}$*, outputs an element* $x \in \mathcal{D}_{\mathsf{td}}$*. We write this as* $x := \mathsf{Inv}_{\mathsf{td}}(y)$*. It is required that for all* (I, td) *output by* $\mathsf{Gen}(1^n)$ *and all* $x \in \mathcal{D}_I$ *we have*

$$\mathsf{Inv}_{\mathsf{td}}(f_I(x)) = x.$$

The final condition means that f_I can be inverted *with* td. The second condition, in contrast, means that f_I cannot be efficiently inverted *without* td. It is immediate that Construction 7.71 can be modified to give a family of trapdoor permutations as long as the RSA problem is hard relative to GenRSA.

We refer to this as the *RSA trapdoor permutation*. Another example of a trapdoor permutation, based on the factoring assumption, will be given in Section 11.2.2.

10.7.2 Public-Key Encryption from Trapdoor Permutations

We now sketch how a public-key encryption scheme can be constructed from an arbitrary family of trapdoor permutations. Although the reader may better appreciate the material in this section after reading Chapter 6, that chapter is not required in order to understand this section. In order to keep this section self-contained, however, some repetition is inevitable.

Before continuing, it will be useful to introduce some shorthand. If the tuple $(\mathsf{Gen}, \mathsf{Samp}, f, \mathsf{Inv})$ is a family of trapdoor permutations and (I, td) is a pair of values output by Gen, we simply write "$x \leftarrow \mathcal{D}_I$" to denote random selection of an element from $\mathcal{D}_I$ (and no longer explicitly refer to algorithm Samp). We also use f_I^{-1} in place of $\mathsf{Inv}_{\mathsf{td}}$, with the understanding that f_I^{-1} can only be efficiently computed if td is known. We will thus refer to (Gen, f) as a family of trapdoor permutations, though formally we still mean $(\mathsf{Gen}, \mathsf{Samp}, f, \mathsf{Inv})$.

Given only I and $f_I(x)$ for a randomly-generated I and a randomly-chosen x, the fact that (Gen_1, f) is a one-way permutation means that we cannot expect to be able to compute x efficiently. However, this does not mean that certain information about x (say, the least-significant bit of x) is hard to compute. The first step in our construction of a public-key encryption scheme will be to *distill* the hardness of a family of trapdoor permutation, by identifying a single bit of information that *is* hard to compute about x. This idea is made concrete in the notion of a *hard-core predicate*. (For those who have studied Chapter 6, the following is the natural adaptation of Definition 6.5 to our context.)

DEFINITION 10.26 *Let $\Pi = (\mathsf{Gen}, f)$ be a family of trapdoor permutations. Let hc be a deterministic polynomial-time algorithm that, on input I and $x \in \mathcal{D}_I$, outputs a single bit $\mathsf{hc}_I(x)$. We say that hc is a* hard-core predicate of Π *if for every probabilistic polynomial-time algorithm $\mathcal{A}$ there exists a negligible function negl such that*

$$\Pr[\mathcal{A}(I, f_I(x)) = \mathsf{hc}_I(x)] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over the experiment in which $\mathsf{Gen}(1^n)$ is run to generate (I, td) and then x is chosen uniformly at random from $\mathcal{D}_I$.

Given a family $\widehat{\Pi}$ of trapdoor permutations and a predicate hc that is hard-core for this family, we can encrypt a single bit in the following way. The receiver runs $\mathsf{Gen}(1^n)$ to generate (I, td), and sets its public key to be I and its private key to be td. Given the public key I, the sender encrypts a single

bit m by: **(1)** choosing a random $x \leftarrow \mathcal{D}_I$; **(2)** computing $y := f_I(x)$; and **(3)** sending the ciphertext $\langle y, \mathsf{hc}_I(x) \oplus m \rangle$. To decrypt the ciphertext $\langle y, m' \rangle$ using private key td, the receiver first computes $x := f_I^{-1}(y)$ using td, and then outputs the message $m := \mathsf{hc}_I(x) \oplus m'$ (see Construction 10.27). It is easy to see that this recovers the original message.

CONSTRUCTION 10.27

Let $\widehat{\Pi} = (\widehat{\mathsf{Gen}}, f)$ be a family of trapdoor permutations, and let hc be a hard-core predicate for $\widehat{\Pi}$. Construct the following public-key encryption scheme:

- Gen: on input 1^n, run $\widehat{\mathsf{Gen}}(1^n)$ to obtain (I, td). Output the public key I and the private key td.
- Enc: on input a public key I and a message $m \in \{0,1\}$, choose a random $x \leftarrow \mathcal{D}_I$ and output the ciphertext

$$\langle f_I(x), \mathsf{hc}_I(x) \oplus m \rangle.$$

- Dec: on input a private key td and a ciphertext $\langle y, m' \rangle$ where $y \in \mathcal{D}_{\mathsf{td}}$, compute $x := f_I^{-1}(y)$ and output the message $\mathsf{hc}_I(x) \oplus m'$.

A public-key encryption scheme from any family of trapdoor permutations.

Intuitively, this is secure since the fact that hc is a hard-core predicate for $\widehat{\Pi}$ exactly implies that $\mathsf{hc}_I(x)$ is *pseudorandom* from the point of view of an eavesdropping adversary who sees the public key I and $f_I(x)$, where x is random. Security follows in a manner analogous to what we are familiar with from the private-key setting. We now prove this formally.

THEOREM 10.28 *If $\widehat{\Pi}$ is a family of trapdoor permutations and hc is a hard-core predicate of $\widehat{\Pi}$, then Construction 10.27 has indistinguishable encryptions under a chosen-plaintext attack.*

PROOF Let Π denote the public-key encryption scheme given by Construction 10.27. As usual, we prove that Π has indistinguishable encryptions in the presence of an eavesdropper, and use Theorem 10.10 to obtain the stated result.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1].$$

We assume, without loss of generality, that the messages m_0, m_1 output by $\mathcal{A}$ are always different. (You should convince yourself that this is indeed without loss of generality.)

Consider the following PPT algorithm $\mathcal{A}_{\mathsf{hc}}$ that attempts to compute $\mathsf{hc}_I(x)$ when given I and $y = f_I(x)$ as input:

Algorithm $\mathcal{A}_{\mathsf{hc}}$:
The algorithm is given I and $y \in \mathcal{D}_I$ as input.

- Set $pk = I$ and run $\mathcal{A}(pk)$ to obtain $m_0, m_1 \in \{0,1\}$.
- Choose independent random bits z and b. Set $m' := m_b \oplus z$.
- Give the ciphertext $\langle y, m' \rangle$ to $\mathcal{A}$ and obtain an output bit b'. If $b' = b$, output z; otherwise, output $\bar{z}$ (the complement of z).

We analyze the behavior of $\mathcal{A}_{\mathsf{hc}}$. Letting x be such that $y = f_I(x)$ (note that x is well-defined since f is a permutation), we can view z as an initial "guess" by $\mathcal{A}_{\mathsf{hc}}$ for the value of $\mathsf{hc}_I(x)$. This guess is correct with probability $1/2$, and incorrect with probability $1/2$. We also have

$$\begin{aligned}&\Pr[\mathcal{A}_{\mathsf{hc}}(I, f_I(x)) = \mathsf{hc}_I(x)] \\ &= \frac{1}{2} \cdot \Big(\Pr[b' = b \mid z = \mathsf{hc}_I(x)] + \Pr[b' \neq b \mid z \neq \mathsf{hc}_I(x)] \Big). \end{aligned} \tag{10.13}$$

When $z = \mathsf{hc}_I(x)$ the view of $\mathcal{A}$ (being run as a sub-routine by $\mathcal{A}_{\mathsf{hc}}$) is distributed identically to $\mathcal{A}$'s view in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ with bit b being used in that experiment. This is true because in this case m' satisfies $m' = m_b \oplus \mathsf{hc}_I(x)$ for a randomly-chosen value of x, and so the ciphertext $\langle y, m' \rangle$ given to $\mathcal{A}$ is indeed a random encryption of m_b. It follows that

$$\Pr[b' = b \mid z = \mathsf{hc}_I(x)] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n).$$

On the other hand, when $z \neq \mathsf{hc}_I(x)$ then the view of $\mathcal{A}$ (being run as a sub-routine by $\mathcal{A}_{\mathsf{hc}}$) is distributed exactly as $\mathcal{A}$'s view in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ *but with bit* $\bar{b}$ *being used in that experiment.* This follows because now m' satisfies

$$m' = m_b \oplus \overline{\mathsf{hc}}_I(x) = m_{\bar{b}} \oplus \mathsf{hc}_I(x),$$

(recalling our assumption that $m_0 \neq m_1$), and so the ciphertext $\langle y, m' \rangle$ given to $\mathcal{A}$ is now a random encryption of $m_{\bar{b}}$. Therefore

$$\Pr[b' = b \mid z \neq \mathsf{hc}_I(x)] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 0] = 1 - \varepsilon(n)$$

and so $\Pr[b' \neq b \mid z \neq \mathsf{hc}_I(x)] = \varepsilon(n)$.

Combining the above with Equation (10.13) we have that

$$\Pr[\mathcal{A}_{\mathsf{hc}}(I, f_I(x)) = \mathsf{hc}_I(x)] = \frac{1}{2} \cdot \Big(\varepsilon(n) + \varepsilon(n) \Big) = \varepsilon(n).$$

Since hc is a hard-core predicate for Π we have $\varepsilon(n) \leq \frac{1}{2} + \mathsf{negl}(n)$. This completes the proof. ∎

It remains only to show that families of trapdoor permutations have hard-core predicates. For some natural families, such as the one based on the RSA assumption that was discussed in the previous section, specific hard-core predicates are known. (As one example, it is known that if the RSA assumption holds then the least-significant bit is hard-core for the RSA family of trapdoor permutations.) For the general case, we can rely on the following result that can be proved by a suitable modification of Theorem 6.6:

THEOREM 10.29 *If a family of trapdoor permutations* Π *exists, then there exists a family of trapdoor permutations* $\widehat{\Pi}$ *along with a predicate* hc *that is hard-core for* $\widehat{\Pi}$.

Encrypting longer messages. Using Proposition 10.11, we know that we can extend Construction 10.27 to encrypt ℓ-bit messages for any polynomial ℓ. Doing so, the ciphertext corresponding to an ℓ-bit message $m = m_1 \cdots m_\ell$ encrypted with respect to the public key I would take the form

$$\left\langle f_I(x_1), \mathsf{hc}_I(x_1) \oplus m_1 \right\rangle, \ldots, \left\langle f_I(x_\ell), \mathsf{hc}_I(x_\ell) \oplus m_\ell \right\rangle$$

with $x_1, \ldots, x_\ell$ chosen independently and uniformly at random from $\mathcal{D}_I$.

We can reduce the size of the ciphertext by having the sender instead proceed as follows: Choose random $x_1 \leftarrow \mathcal{D}_I$ and compute $x_{i+1} := f_I(x_i)$ for $i = 1$ to ℓ. Then output the ciphertext

$$\left\langle x_{\ell+1}, \mathsf{hc}_I(x_1) \oplus m_1, \cdots, \mathsf{hc}_I(x_\ell) \oplus m_\ell \right\rangle.$$

A proof that this is secure uses ideas from Section 6.4, and is left as an advanced exercise.

References and Additional Reading

The idea of public-key encryption was first proposed (in the open literature, at least) by Diffie and Hellman [47]. Somewhat amazingly, the El Gamal encryption scheme [59] was not proposed until 1984 even though it can be viewed as a direct transformation of the Diffie-Hellman key exchange protocol introduced by Diffie and Hellman in 1976 (see Exercise 10.4). Rivest, Shamir, and Adleman [122] introduced the RSA assumption and proposed a public-key encryption scheme based on this assumption.

Definition 10.3 is rooted in the seminal work of Goldwasser and Micali [69], who were also the first to recognize the necessity of *probabilistic* encryption for satisfying this definition. A proof of security for a special case of hybrid encryption was first given by Blum and Goldwasser [22].

The public-key encryption scheme suggested by Rivest, Shamir, and Adleman [122] corresponds to the textbook RSA scheme shown here. The attacks described in Section 10.4.2 are due to [73, 45, 133, 27]; see [99, Chapter 8] and [25] for additional attacks and further explanation. The *PKCS #1 RSA Cryptography Standard* (both the latest version and previous versions) is available for download from `http://www.rsa.com/rsalabs`. A proof of Theorem 10.19 can be derived from results in [5, 75].

As noted in Chapter 4, chosen-ciphertext attacks were first formally defined by Naor and Yung [107] and Rackoff and Simon [121]. The chosen-ciphertext attacks on the "textbook RSA" and El Gamal schemes are immediate; the attack on PKCS #1 v1.5 is due to Bleichenbacher [20]. For more recent definitional treatments of public-key encryption under stronger attacks, including chosen-ciphertext attacks, the reader is referred to the works of Dolev et al. [50] and Bellare et al. [10]. The first efficient public-key encryption scheme secure against chosen-ciphertext attack was shown by Cramer and Shoup [39]. The expository article by Shoup [129] contains a discussion of the importance of security against chosen-ciphertext attacks.

The existence of public-key encryption based on arbitrary trapdoor permutations was shown by Yao [149], and the efficiency improvement discussed at the end of Section 10.7.2 is due to Blum and Goldwasser [22]. The reader interested in finding out more about hard-core predicates for the RSA family of trapdoor permutations is invited to peruse [5, 75, 4] and references therein.

When using any encryption scheme in practice the question of what key-length to use arises. This issue should not be taken lightly, and we refer the reader to [94] for a treatment of this issue.

Exercises

10.1 Assume a public-key encryption scheme for single-bit messages. Show that, given pk and a ciphertext c computed via $c \leftarrow \mathsf{Enc}_{pk}(m)$, it is possible for an unbounded adversary to determine m with probability 1. This shows that perfectly-secret public-key encryption is impossible.

10.2 Say a deterministic public-key encryption scheme is used to encrypt a message m that is known to lie in a small set of $\mathcal{L}$ possible values. Show how it is possible to determine m in time linear in $\mathcal{L}$ (assume that encryption of an element takes a single unit of time).

10.3 Show that for any CPA-secure public-key encryption scheme, the size of the ciphertext after encrypting a single bit is superlogarithmic in the security parameter. (That is, for $(pk, sk) \leftarrow \mathsf{Gen}(1^n)$ it must hold that $|\mathsf{Enc}_{pk}(b)| = \omega(\log n)$ for any $b \in \{0,1\}$.)

Hint: If not, the range of possible ciphertexts is only polynomial in size.

10.4 Show that any 2-round key-exchange protocol (that is, where each party sends a single message) satisfying Definition 9.1 can be converted into a public-key encryption scheme that is CPA-secure.

10.5 Show that in Definition 10.7, we can assume without loss of generality that $\mathcal{A}$ always outputs two vectors containing *exactly* $t(n)$ messages each. That is, show how to construct, for any scheme Π and any adversary $\mathcal{A}$, an adversary $\mathcal{A}'$ that always outputs vectors of the same length $t(n)$ for each fixed value of n and such that

$$\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A}',\Pi}(n) = 1].$$

10.6 Prove Claim 10.9.

10.7 Fix N, and assume there exists an adversary $\mathcal{A}$ running in time t for which

$$\Pr\left[\mathcal{A}\left([x^e \bmod N]\right) = x\right] = 0.01,$$

where the probability is taken over random choice of $x \leftarrow \mathbb{Z}^*_N$. Show that it is possible to construct an adversary $\mathcal{A}'$ for which

$$\Pr\left[\mathcal{A}\left([x^e \bmod N]\right) = x\right] = 0.99.$$

The running time t' of $\mathcal{A}'$ should satisfy $t' = \mathsf{poly}(\|N\|, t)$.

Hint: Use the fact that $y^{1/e} \cdot r = (y \cdot r^e)^{1/e} \bmod N$.

10.8 The public exponent e in RSA can be chosen arbitrarily, subject to $\gcd(e, \phi(N)) = 1$. Popular choices of e include $e = 3$ and $e = 2^{16} + 1$. Explain why such e are preferable to a random value of the same length.

Hint: See the algorithm for modular exponentiation in Appendix B.2.3.

10.9 Let GenRSA have the usual meaning. Consider the following experiment for an algorithm $\mathcal{A}$ and a function ℓ with $\ell(n) \leq 2n - 2$ for all n:

The padded RSA experiment $\mathsf{PAD}_{\mathcal{A},\mathsf{GenRSA},\ell}(n)$:

(a) Run $\mathsf{GenRSA}(1^n)$ to obtain output (N, e, d).

(b) Give N, e to $\mathcal{A}$, who outputs a string $m \in \{0,1\}^{\ell(n)}$.

(c) Choose random $y_0 \leftarrow \mathbb{Z}^*_N$. Choose random $r \leftarrow \{0,1\}^{\|N\| - \ell(n) - 1}$ and set

$$y_1 := [(r \| m)^e \bmod N].$$

(d) Choose random $b \leftarrow \{0,1\}$. Adversary $\mathcal{A}$ is given y_b, and outputs a bit b'.

(e) The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

Say the ℓ-padded RSA problem is hard relative to GenRSA if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function negl such that $\Pr[\mathsf{PAD}_{\mathcal{A},\mathsf{GenRSA},\ell}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$.

Prove that if the ℓ-padded RSA problem is hard relative to GenRSA, then the padded RSA encryption scheme (Construction 10.18) using ℓ is CPA-secure.

10.10 Say a function f is *hard-core for* GenRSA if for all PPT algorithms $\mathcal{A}$ there exists a negligible function negl such that

$$\Big|\Pr[\mathcal{A}(N, e, y, f(x)) = 1] - \Pr[\mathcal{A}(N, e, y, f(r)) = 1]\Big| \leq \mathsf{negl}(n),$$

where in each case the probabilities are taken over the experiment in which $\mathsf{GenRSA}(1^n)$ outputs (N, e, d), random $x, r \leftarrow \mathbb{Z}_N^*$ are chosen, and y is set equal to $[x^e \bmod N]$.

For $x \in \mathbb{Z}_N^*$, let $\mathsf{lsb}(x)$ (resp., $\mathsf{msb}(x)$) denote the least- (resp., most-) significant bit of x when written as an integer using exactly $\|N\|$ bits. Define $f(x) \stackrel{\text{def}}{=} \mathsf{msb}(x)\|\mathsf{lsb}(x)$. It can be shown that if the RSA problem is hard relative to GenRSA, then f is hard-core for GenRSA [4]. Prove Theorem 10.19 by relying on this result.

Hint: Note that in Construction 10.18, $\mathsf{msb}(r\|m)$ is always equal to 0.

10.11 Consider the following public-key encryption scheme. The public key is $(\mathbb{G}, q, g, h)$ and the private key is x, generated exactly as in the El Gamal encryption scheme. In order to encrypt a bit b, the sender does the following:

(a) If $b = 0$ then choose a random $y \leftarrow \mathbb{Z}_q$ and compute $c_1 = g^y$ and $c_2 = h^y$. The ciphertext is $\langle c_1, c_2 \rangle$.

(b) If $b = 1$ then choose independent random $y, z \leftarrow \mathbb{Z}_q$, compute $c_1 = g^y$ and $c_2 = g^z$, and set the ciphertext equal to $\langle c_1, c_2 \rangle$.

Show that it is possible to decrypt efficiently given knowledge of x. Prove that this encryption scheme is CPA-secure if the decisional Diffie-Hellman problem is hard relative to $\mathcal{G}$.

10.12 The natural way of applying hybrid encryption to the El Gamal encryption scheme is as follows. The public key is $pk = \langle \mathbb{G}, q, g, h \rangle$ as in the El Gamal scheme, and to encrypt a message m the sender chooses random $k \leftarrow \{0,1\}^n$ and sends

$$\langle g^r,\ h^r \cdot k,\ \mathsf{Enc}_k(m) \rangle,$$

where $r \leftarrow \mathbb{Z}_q$ is chosen at random and Enc represents a private-key encryption scheme. Suggest an improvement that results in a shorter ciphertext containing only a *single* group element followed by a private-key encryption of m.

10.13 Show that Proposition 10.11 does not hold in the setting of CCA-security. In contrast, Theorem 10.10 *does* hold in the setting of CCA-security. Explain why in the setting of CPA security there is no distinction between multiple messages and a single long message, whereas in the setting of CCA security there is.

10.14 Consider the following version of padded RSA encryption. Assume that the message m to be encrypted has length $\|N\|/2$ (i.e., roughly half the length of the modulus). To encrypt, first pad m to the left with one byte of zeroes, then 10 random bytes, and then all zeroes; the result is denoted $\bar{m}$ (that is, $\bar{m} = (0^k \parallel r \parallel 00000000 \parallel m)$, where k is the number of zeroes needed to make $\bar{m}$ the appropriate size). Finally, compute $c = [\bar{m}^e \bmod N]$. Describe a chosen-ciphertext attack on this scheme. Why is it easier to construct a chosen-ciphertext attack on this scheme than on PKCS #1 v1.5?

10.15 Let Π be a CCA-secure public-key encryption scheme and let Π' be a CCA-secure private-key encryption scheme. Is Construction 10.12 instantiated using Π and Π' CCA-secure? Prove your answer.

10.16 Consider the following variant of Construction 10.27 which gives an alternative way of encrypting using any family of trapdoor permutations:

CONSTRUCTION 10.30

Let $\widehat{\Pi} = (\widehat{\mathsf{Gen}}, f)$ and hc be as in Construction 10.27.

- Gen: as in Construction 10.27.
- Enc: on input a public key I and a message $m \in \{0,1\}$, choose a random $x \leftarrow \mathcal{D}_I$ such that $\mathsf{hc}_I(x) = m$, and output the ciphertext $f_I(x)$.
- Dec: on input a private key td and a ciphertext y with $y \in \mathcal{D}_{\mathsf{td}}$, compute $x := f_I^{-1}(y)$ and output the message $\mathsf{hc}_I(x)$.

(a) Argue that encryption can be performed in polynomial time.

(b) Prove that if $\widehat{\Pi}$ is a family of trapdoor permutations and hc is a hard-core predicate of $\widehat{\Pi}$, then this construction is CPA-secure.

10.17 Consider the following protocol for two parties A and B to flip a fair coin (more complicated versions of this might be used for Internet gambling): **(1)** a trusted party T publishes her public key pk; **(2)** A chooses a random bit b_A, encrypts it using pk, and announces the ciphertext c_A to B and T; **(3)** next, B acts symmetrically and announces a ciphertext $c_B \neq c_A$; **(4)** T decrypts both c_A and c_B, and the parties XOR the results to obtain the value of the coin.

(a) Argue that even if A is dishonest (but B is honest), the final value of the coin is uniformly distributed.

(b) Assume the parties use El Gamal encryption (where the bit b is encoded as the group element g^b). Show how a dishonest B can bias the coin to any value he likes.

(c) Suggest what type of encryption scheme would be appropriate to use here. Can you define an appropriate notion of security and prove that your suggestion achieves this definition?

Chapter 11

* Additional Public-Key Encryption Schemes

In the previous chapter we saw some examples of public-key encryption schemes based on the RSA and decisional Diffie-Hellman problems. Here, we explore additional encryption schemes based on other number-theoretic problems related to the hardness of factoring. The schemes discussed in this chapter are not fundamentally any less important than the schemes covered in the previous chapter — although RSA and El Gamal encryption *are* more widely used than any of the schemes discussed here — rather, the schemes we discuss in this chapter were placed here because they require a bit more number theory than we have covered to this point. Each of the schemes we show here is well worth understanding, at least from a theoretical standpoint:

- The *Goldwasser-Micali encryption scheme*, based on the hardness of distinguishing quadratic residues from (certain) quadratic non-residues modulo a composite, was the first scheme proven to be CPA-secure. The cryptographic assumption on which the scheme relies is also useful in other contexts.
- The *Rabin encryption scheme* is very similar to the RSA encryption scheme, but with one crucial difference: it is possible to prove that the Rabin encryption scheme is CPA-secure under the assumption that factoring is hard. Recall that, in contrast, hardness of the RSA problem (and thus the security of any encryption scheme based on RSA) is not known to follow from the factoring assumption.
- The *Paillier encryption scheme* is based on a cryptographic assumption related (but not known to be identical) to factoring. In contrast to Goldwasser-Micali encryption or the provably-secure variant of Rabin encryption, in Paillier encryption the message is not encrypted bit-by-bit; thus, the efficiency of Paillier encryption is comparable to that of El Gamal encryption. The Paillier encryption scheme also has other advantages (like the fact that it is homomorphic) that we will discuss in Section 11.3.

Throughout this chapter, we let p and q denote odd primes, and let N denote a product of two distinct odd primes. We will use the Chinese remainder theorem and thus assume familiarity with Section 7.1.5.

11.1 The Goldwasser-Micali Encryption Scheme

We begin with a discussion of the Goldwasser-Micali encryption scheme. Before we can present the scheme, we need to develop a better understanding of *quadratic residues*. We first explore the easier case of quadratic residues modulo a prime p, and then look at the slightly more complicated case of quadratic residues modulo a composite N.

11.1.1 Quadratic Residues Modulo a Prime

Given a group $\mathbb{G}$, an element $y \in \mathbb{G}$ is a *quadratic residue* if there exists an $x \in \mathbb{G}$ with $x^2 = y$. In this case, we call x a *square root* of y. An element that is not a quadratic residue is called a *quadratic non-residue*. In an abelian group, the set of quadratic residues forms a subgroup.

In the specific case of $\mathbb{Z}_p^*$, we have that y is a quadratic residue if there exists an x with $x^2 = y \bmod p$. We begin with an easy observation.

PROPOSITION 11.1 *Let $p > 2$ be prime. Every quadratic residue in $\mathbb{Z}_p^*$ has exactly two square roots.*

PROOF Let $y \in \mathbb{Z}_p^*$ be a quadratic residue. Then there exists an $x \in \mathbb{Z}_p^*$ such that $x^2 = y \bmod p$. Clearly, $(-x)^2 = x^2 = y \bmod p$. Furthermore, $-x \neq x \bmod p$: if $-x = x \bmod p$ then $2x = 0 \bmod p$ which implies $p \,|\, 2x$. Since p is prime, this would mean that either $p \,|\, 2$ (which is impossible since $p > 2$) or $p \,|\, x$ (which is impossible since $0 < x < p$). So, $[x \bmod p]$ and $[-x \bmod p]$ are distinct elements of $\mathbb{Z}_p^*$, and y has at least two square roots.

Let $x' \in \mathbb{Z}_p^*$ be a square root of y. Then $x^2 = y = (x')^2 \bmod p$ implying that $x^2 - (x')^2 = 0 \bmod p$. Factoring the left-hand side we obtain

$$(x - x')(x + x') = 0 \bmod p\,,$$

so that either $p \,|\, (x - x')$ or $p \,|\, (x + x')$. In the first case, $x' = x \bmod p$ and in the second case $x' = -x \bmod p$, showing that y indeed has only $[\pm x \bmod p]$ as square roots. ∎

Let $\mathsf{sq}_p : \mathbb{Z}_p^* \to \mathbb{Z}_p^*$ be the function $\mathsf{sq}_p(x) \stackrel{\text{def}}{=} [x^2 \bmod p]$. The above proposition shows that sq_p is a two-to-one function when $p > 2$ is prime. This immediately implies that *exactly half the elements of $\mathbb{Z}_p^*$ are quadratic residues*. We denote the set of quadratic residues modulo p by $\mathcal{QR}_p$, and the set of quadratic non-residues by $\mathcal{QNR}_p$. We have just seen that for $p > 2$ prime

$$|\mathcal{QR}_p| = |\mathcal{QNR}_p| = \frac{|\mathbb{Z}_p^*|}{2} = \frac{p-1}{2}\,.$$

Define $\mathcal{J}_p(x)$, the *Jacobi symbol of x modulo p*, as follows.[1] Let $p > 2$ be prime, and $x \in \mathbb{Z}_p^*$. Then

$$\mathcal{J}_p(x) \stackrel{\text{def}}{=} \begin{cases} +1 \text{ if } x \text{ is a quadratic residue modulo } p \\ -1 \text{ if } x \text{ is } \textit{not} \text{ a quadratic residue modulo } p \end{cases}.$$

The notation can be extended in the natural way for any x relatively prime to p by defining $\mathcal{J}_p(x) = \mathcal{J}_p([x \bmod p])$.

Can we characterize the quadratic residues in $\mathbb{Z}_p^*$ for $p > 2$ prime? We begin with the fact that $\mathbb{Z}_p^*$ is a cyclic group of order $p-1$ (see Theorem 7.53). Let g be a generator of $\mathbb{Z}_p^*$. This means that

$$\mathbb{Z}_p^* = \{g^0, g^1, g^2, \ldots, g^{\frac{p-1}{2}-1}, g^{\frac{p-1}{2}}, g^{\frac{p-1}{2}+1}, \ldots, g^{p-2}\}$$

(recall that p is odd, so $p-1$ is even). Squaring each element in this list and reducing modulo $p-1$ in the exponent (cf. Corollary 7.15) yields a list of all the quadratic residues in $\mathbb{Z}_p^*$:

$$\mathcal{QR}_p = \{g^0, g^2, g^4, \ldots, g^{p-3}, g^0, g^2, \ldots, g^{p-3}\}.$$

(Note that each quadratic residue appears twice in this list.) We see that the quadratic residues in $\mathbb{Z}_p^*$ are exactly those elements that can be written as g^i with $i \in \{0, \ldots, p-2\}$ an *even* integer.

The above characterization leads to a simple way to compute the Jacobi symbol and thus tell whether a given element $x \in \mathbb{Z}_p^*$ is a quadratic residue or not.

PROPOSITION 11.2 *Let $p > 2$ be a prime. Then $\mathcal{J}_p(x) = x^{\frac{p-1}{2}} \bmod p$.*

PROOF Let g be an arbitrary generator of $\mathbb{Z}_p^*$. If x is a quadratic residue modulo p, our earlier discussion shows that $x = g^i$ for some even integer i. Writing $i = 2j$ with j an integer we then have

$$x^{\frac{p-1}{2}} = \left(g^{2j}\right)^{\frac{p-1}{2}} = g^{(p-1)j} = \left(g^{p-1}\right)^j = 1^j = 1 \bmod p\,,$$

and so $x^{\frac{p-1}{2}} = +1 = \mathcal{J}_p(x) \bmod p$ as claimed.

On the other hand, if x is not a quadratic residue then $x = g^i$ for some odd integer i. Writing $i = 2j+1$ with j an integer we have

$$x^{\frac{p-1}{2}} = \left(g^{2j+1}\right)^{\frac{p-1}{2}} = \left(g^{2j}\right)^{\frac{p-1}{2}} \cdot g^{\frac{p-1}{2}} = 1 \cdot g^{\frac{p-1}{2}} = g^{\frac{p-1}{2}} \bmod p.$$

[1] $\mathcal{J}_p(x)$ is also sometimes called the *Legendre symbol* of x, and denoted by $\mathcal{L}_p(x)$; we have chosen our notation to be consistent with notation introduced later.

Now,

$$\left(g^{\frac{p-1}{2}}\right)^2 = g^{p-1} = 1 \bmod p,$$

and so $g^{\frac{p-1}{2}} = \pm 1 \bmod p$ since $[\pm 1 \bmod p\,]$ are the two square roots of 1 (cf. Proposition 11.1). Since g is a generator, it has order $p-1$ and so $g^{\frac{p-1}{2}} \neq 1 \bmod p$. It follows that $x^{\frac{p-1}{2}} = -1 = \mathcal{J}_p(x) \bmod p$. ■

Proposition 11.2 directly gives a polynomial-time algorithm for testing whether a given element $x \in \mathbb{Z}_p^*$ is a quadratic residue or not.

ALGORITHM 11.3
Deciding quadratic residuosity modulo a prime

Input: A prime p; element $x \in \mathbb{Z}_p^*$
Output: $\mathcal{J}_p(x)$ (or, equivalently, whether x is a quadratic residue or quadratic non-residue)

$b := \left[x^{\frac{p-1}{2}} \bmod p\right]$
if $b = 1$ **return** "quadratic residue"
else return "quadratic non-residue"

We conclude this section by noting a nice multiplicative property of quadratic residues and non-residues modulo p.

PROPOSITION 11.4 *Let $p > 2$ be a prime, and $x, y \in \mathbb{Z}_p^*$. Then*

$$\mathcal{J}_p(xy) = \mathcal{J}_p(x) \cdot \mathcal{J}_p(y).$$

PROOF Using the previous proposition,

$$\mathcal{J}_p(xy) = (xy)^{\frac{p-1}{2}} = x^{\frac{p-1}{2}} \cdot y^{\frac{p-1}{2}} = \mathcal{J}_p(x) \cdot \mathcal{J}_p(y) \bmod p.$$

Since $\mathcal{J}_p(xy), \mathcal{J}_p(x), \mathcal{J}_p(y) = \pm 1$, equality holds over the integers as well. ■

COROLLARY 11.5 *Let $p > 2$ be prime, $x, x' \in \mathcal{QR}_p$, and $y, y' \in \mathcal{QNR}_p$. Then:*

1. $[xx' \bmod p\,] \in \mathcal{QR}_p$.
2. $[yy' \bmod p\,] \in \mathcal{QR}_p$.
3. $[xy \bmod p\,] \in \mathcal{QNR}_p$.

11.1.2 Quadratic Residues Modulo a Composite

We now turn our attention to quadratic residues in the group $\mathbb{Z}_N^*$. Characterizing the quadratic residues modulo N is easy if we use the results of the previous section in conjunction with the Chinese remainder theorem. Recall that the Chinese remainder theorem says that $\mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*$, and we let $y \leftrightarrow (y_p, y_q)$ denote the correspondence guaranteed by the theorem (i.e., $y_p = [y \bmod p\,]$ and $y_q = [y \bmod q\,]$). The key observation is:

PROPOSITION 11.6 *Let $N = pq$ with p, q distinct primes, and $y \in \mathbb{Z}_N^*$ with $y \leftrightarrow (y_p, y_q)$. Then y is a quadratic residue modulo N if and only if y_p is a quadratic residue modulo p and y_q is a quadratic residue modulo q.*

PROOF If y is a quadratic residue modulo N then, by definition, there exists an $x \in \mathbb{Z}_N^*$ such that $x^2 = y \bmod N$. Let $x \leftrightarrow (x_p, x_q)$. Then

$$(y_p,\, y_q) \leftrightarrow y = x^2 \leftrightarrow (x_p,\, x_q)^2 = ([x_p^2 \bmod p\,],\, [x_q^2 \bmod q\,]),$$

where $(x_p, x_q)^2$ is simply the square of element (x_p, x_q) in the group $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$. We have thus shown that

$$y_p = x_p^2 \bmod p \quad \text{and} \quad y_q = x_q^2 \bmod q \tag{11.1}$$

and y_p, y_q are quadratic residues (with respect to the appropriate moduli).

Conversely, if $y \leftrightarrow (y_p, y_q)$ and y_p, y_q are quadratic residues modulo p and q, respectively, then there exist $x_p \in \mathbb{Z}_p^*$ and $x_q \in \mathbb{Z}_q^*$ such that Equation (11.1) holds. Let $x \in \mathbb{Z}_N^*$ be such that $x \leftrightarrow (x_p, x_q)$. Reversing the above steps shows that x is a square root of y modulo N. ∎

The above proposition characterizes the quadratic residues modulo N. A careful examination of the proof yields another important observation: each quadratic residue $y \in \mathbb{Z}_N^*$ has exactly *four* square roots. To see this, let $y \leftrightarrow (y_p, y_q)$ be a quadratic residue modulo N and let x_p, x_q be square roots of y_p and y_q modulo p and q, respectively. Then the four square roots of y are given by the elements in $\mathbb{Z}_N^*$ corresponding to

$$(x_p,\, x_q), \quad (-x_p,\, x_q), \quad (x_p,\, -x_q), \quad (-x_p,\, -x_q). \tag{11.2}$$

Each of these is a square root of y since

$$\begin{aligned}(\pm x_p,\, \pm x_q)^2 &= \Big([(\pm x_p)^2 \bmod p\,],\, [(\pm x_q)^2 \bmod q\,]\Big) \\ &= ([x_p^2 \bmod p\,],\, [x_q^2 \bmod q\,]) = (y_p,\, y_q) \leftrightarrow y\end{aligned}$$

(where again the notation $(\cdot,\cdot)^2$ refers to squaring in the group $\mathbb{Z}_p \times \mathbb{Z}_q$). The Chinese remainder theorem guarantees that the four elements in Equation (11.2) each correspond to *distinct* elements of $\mathbb{Z}_N^*$, since x_p and $-x_p$ are unique modulo p (and similarly for x_q and $-x_q$ modulo q).

Example 11.7
Consider $\mathbb{Z}_{15}^*$ (the correspondence given by the Chinese remainder theorem is tabulated in Example 7.25). Element 4 is a quadratic residue modulo 15 with square root 2. Since $2 \leftrightarrow (2,2)$, the other square roots of 4 are given by:

- $\big(2, [-2 \bmod 3]\big) = (2,1) \leftrightarrow 7$;
- $\big([-2 \bmod 5], 2\big) = (3,2) \leftrightarrow 8$; and
- $\big([-2 \bmod 5], [-2 \bmod 3]\big) = (3,1) \leftrightarrow 13$.

One can verify that $7^2 = 8^2 = 13^2 = 4 \bmod 15$. ♢

Let $\mathcal{QR}_N$ denote the set of quadratic residues modulo N. Since squaring modulo N is a four-to-one function, we immediately see that exactly 1/4 of the elements of $\mathbb{Z}_N^*$ are quadratic residues. Alternately, we could note that since $y \in \mathbb{Z}_N^*$ is a quadratic residue if and only if y_p, y_q are quadratic residues, there is a one-to-one correspondence between $\mathcal{QR}_N$ and $\mathcal{QR}_p \times \mathcal{QR}_q$. Thus, the fraction of quadratic residues modulo N is

$$\frac{|\mathcal{QR}_N|}{|\mathbb{Z}_N^*|} = \frac{|\mathcal{QR}_p| \cdot |\mathcal{QR}_q|}{|\mathbb{Z}_N^*|} = \frac{\frac{p-1}{2} \cdot \frac{q-1}{2}}{(p-1)(q-1)} = \frac{1}{4},$$

in agreement with the above.

In the previous section, we defined the Jacobi symbol $\mathcal{J}_p(x)$ for $p > 2$ prime. We extend the definition to the case of $N = pq$ a product of distinct, odd primes as follows. For any x relatively prime to $N = pq$,

$$\begin{aligned} \mathcal{J}_N(x) &\stackrel{\text{def}}{=} \mathcal{J}_p(x) \cdot \mathcal{J}_q(x) \\ &= \mathcal{J}_p([x \bmod p\,]) \cdot \mathcal{J}_q([x \bmod q\,]). \end{aligned}$$

We define $\mathcal{J}_N^{+1}$ as the set of elements in $\mathbb{Z}_N^*$ having Jacobi symbol $+1$, and define $\mathcal{J}_N^{-1}$ analogously.

We know from Proposition 11.6 that if x is a quadratic residue modulo N, then $[x \bmod p\,]$ and $[x \bmod q\,]$ are quadratic residues modulo p and q, respectively; that is, $\mathcal{J}_p(x) = \mathcal{J}_q(x) = +1$. So $\mathcal{J}_N(x) = +1$ and we see that:

If x is a quadratic residue modulo N, then $\mathcal{J}_N(x) = +1$.

However, $\mathcal{J}_N(x) = +1$ can also occur when $\mathcal{J}_p(x) = \mathcal{J}_q(x) = -1$; that is, when *both* $[x \bmod p\,]$ and $[x \bmod q\,]$ are *not* quadratic residues modulo p and q (and so x is not a quadratic residue modulo N). This turns out to be useful for the Goldwasser-Micali encryption scheme, and we therefore introduce the notation $\mathcal{QNR}_N^{+1}$ for the set of elements of this type. That is

$$\mathcal{QNR}_N^{+1} \stackrel{\text{def}}{=} \left\{ x \in \mathbb{Z}_N^* \;\middle|\; \begin{array}{c} x \text{ is not a quadratic residue modulo } N, \\ \text{but } \mathcal{J}_N(x) = +1 \end{array} \right\}.$$

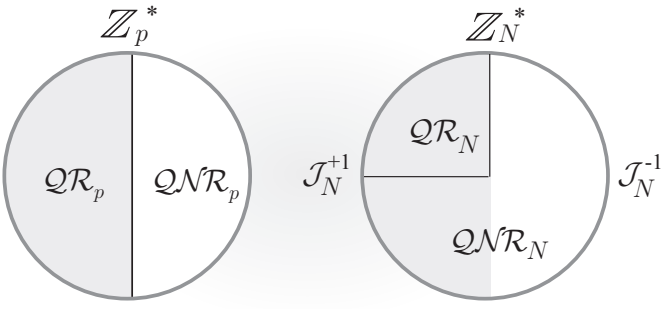

FIGURE 11.1: The structure of $\mathbb{Z}_p^*$ and $\mathbb{Z}_N^*$.

It is now easy to prove the following (see Figure 11.1):

PROPOSITION 11.8 *Let $N = pq$ with p, q distinct, odd primes. Then:*

1. *Exactly half the elements of $\mathbb{Z}_N^*$ are in $\mathcal{J}_N^{+1}$.*
2. *$\mathcal{QR}_N$ is contained in $\mathcal{J}_N^{+1}$.*
3. *Exactly half the elements of $\mathcal{J}_N^{+1}$ are in $\mathcal{QR}_N$ (the other half are in $\mathcal{QNR}_N^{+1}$).*

PROOF We know that $\mathcal{J}_N(x) = +1$ if either $\mathcal{J}_p(x) = \mathcal{J}_q(x) = +1$ or $\mathcal{J}_p(x) = \mathcal{J}_q(x) = -1$. We also know (from the previous section) that exactly half the elements of $\mathbb{Z}_p^*$ have Jacobi symbol $+1$, and half have Jacobi symbol -1 (and similarly for $\mathbb{Z}_q^*$). Defining $\mathcal{J}_p^{+1}$, $\mathcal{J}_p^{-1}$, $\mathcal{J}_q^{+1}$, and $\mathcal{J}_q^{-1}$ in the natural way, we thus have:

$$\begin{aligned}\left|\mathcal{J}_N^{+1}\right| &= |\mathcal{J}_p^{+1} \times \mathcal{J}_q^{+1}| + |\mathcal{J}_p^{-1} \times \mathcal{J}_q^{-1}| \\ &= |\mathcal{J}_p^{+1}| \cdot |\mathcal{J}_q^{+1}| + |\mathcal{J}_p^{-1}| \cdot |\mathcal{J}_q^{-1}| \\ &= \frac{(p-1)}{2}\frac{(q-1)}{2} + \frac{(p-1)}{2}\frac{(q-1)}{2} = \frac{\phi(N)}{2}.\end{aligned}$$

So $\left|\mathcal{J}_N^{+1}\right| = |\mathbb{Z}_N^*|/2$, proving that half the elements of $\mathbb{Z}_N^*$ are in $\mathcal{J}_N^{+1}$.

We have noted earlier that all quadratic residues modulo N have Jacobi symbol $+1$, showing that $\mathcal{QR}_N \subseteq \mathcal{J}_N^{+1}$.

Since $x \in \mathcal{QR}_N$ if and only if $\mathcal{J}_p(x) = \mathcal{J}_q(x) = +1$, we have

$$|\mathcal{QR}_N| = |\mathcal{J}_p^{+1} \times \mathcal{J}_q^{+1}| = \frac{(p-1)}{2}\frac{(q-1)}{2} = \frac{\phi(N)}{4},$$

and so $|\mathcal{QR}_N| = \left|\mathcal{J}_N^{+1}\right|/2$. Since $\mathcal{QR}_N$ is a subset of $\mathcal{J}_N^{+1}$, this proves that half the elements of $\mathcal{J}_N^{+1}$ are in $\mathcal{QR}_N$. ■

The next two results are analogues of Proposition 11.4 and Corollary 11.5.

PROPOSITION 11.9 *Let $N = pq$ be a product of distinct, odd primes, and $x, y \in \mathbb{Z}_N^*$. Then $\mathcal{J}_N(xy) = \mathcal{J}_N(x) \cdot \mathcal{J}_N(y)$.*

PROOF Using the definition of $\mathcal{J}_N(\cdot)$ and Proposition 11.4, we have

$$\begin{aligned}\mathcal{J}_N(xy) = \mathcal{J}_p(xy) \cdot \mathcal{J}_q(xy) &= \mathcal{J}_p(x) \cdot \mathcal{J}_p(y) \cdot \mathcal{J}_q(x) \cdot \mathcal{J}_q(y) \\ &= \mathcal{J}_p(x) \cdot \mathcal{J}_q(x) \cdot \mathcal{J}_p(y) \cdot \mathcal{J}_q(y) = \mathcal{J}_N(x) \cdot \mathcal{J}_N(y).\end{aligned}$$

■

COROLLARY 11.10 *Let $N = pq$ be a product of distinct, odd primes, and say $x, x' \in \mathcal{QR}_N$ and $y, y' \in \mathcal{QNR}_N^{+1}$. Then:*

1. $[xx' \bmod N] \in \mathcal{QR}_N$.
2. $[yy' \bmod N] \in \mathcal{QR}_N$.
3. $[xy \bmod N] \in \mathcal{QNR}_N^{+1}$.

PROOF We prove the final claim; proofs of the others are similar. Since $x \in \mathcal{QR}_N$, we have $\mathcal{J}_p(x) = \mathcal{J}_q(x) = +1$. Since $y \in \mathcal{QNR}_N^{+1}$, we have $\mathcal{J}_p(y) = \mathcal{J}_q(y) = -1$. Using Proposition 11.4,

$$\mathcal{J}_p(xy) = \mathcal{J}_p(x) \cdot \mathcal{J}_p(y) = -1 \quad \text{and} \quad \mathcal{J}_q(xy) = \mathcal{J}_q(x) \cdot \mathcal{J}_q(y) = -1,$$

and so $\mathcal{J}_N(xy) = +1$. But xy is not a quadratic residue modulo N, since $\mathcal{J}_p(xy) = -1$ and so $[xy \bmod p]$ is not a quadratic residue modulo p. We conclude that $xy \in \mathcal{QNR}_N^{+1}$. ■

In contrast to Corollary 11.5, it is *not* true that $y, y' \in \mathcal{QNR}_N$ implies $yy' \in \mathcal{QR}_N$. (Instead, as indicated in the corollary, this is only guaranteed if $y, y' \in \mathcal{QNR}_N^{+1}$.) For example, we could have $\mathcal{J}_p(y) = +1$, $\mathcal{J}_q(y) = -1$ and $\mathcal{J}_p(y') = -1$, $\mathcal{J}_q(y') = +1$, so $\mathcal{J}_p(yy') = \mathcal{J}_q(yy') = -1$ and yy' is not a quadratic residue even though $\mathcal{J}_N(yy') = +1$.

11.1.3 The Quadratic Residuosity Assumption

In Section 11.1.1, we showed an efficient algorithm for deciding whether a given input x is a quadratic residue modulo a prime p. Can we adapt the algorithm to work modulo a composite number N? Proposition 11.6 gives an easy solution to this problem *provided the factorization of N is known.* This is written in algorithmic form below in Algorithm 11.11.

ALGORITHM 11.11
Deciding quadratic residuosity modulo a composite of known factorization

Input: Composite $N = pq$; the factors p and q; element $x \in \mathbb{Z}_N^*$
Output: A decision as to whether $x \in \mathcal{QR}_N$

compute $\mathcal{J}_p(x)$ and $\mathcal{J}_q(x)$
if $\mathcal{J}_p(x) = \mathcal{J}_q(x) = +1$ **return** "quadratic residue"
else return "quadratic non-residue"

(As always, we assume the factors of N are distinct odd primes.) A simple modification of the above algorithm allows for computing $\mathcal{J}_N(x)$ when the factorization of N is known.

When the factorization of N is *unknown*, however, there is no known polynomial-time algorithm for deciding whether a given x is a quadratic residue modulo N or not. Somewhat surprisingly, a polynomial-time algorithm *is* known for computing $\mathcal{J}_N(x)$ without the factorization of N. (Although the algorithm itself is not that complicated, its proof of correctness is beyond the scope of this book and we therefore do not present the algorithm at all. The interested reader can refer to the references listed at the end of this chapter.) This leads to a partial test of quadratic residuosity: if, for a given input x, it holds that $\mathcal{J}_N(x) = -1$, then x cannot possibly be a quadratic residue. (See Proposition 11.8.) This test says nothing in case $\mathcal{J}_N(x) = +1$, and it is widely believed that there does *not* exist any polynomial-time algorithm for deciding quadratic residuosity in this case (that performs better than random guessing).

We now formalize this assumption. Let $\mathsf{GenModulus}$ be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$, and p and q are n-bit primes except with probability negligible in n.

DEFINITION 11.12 *We say* deciding quadratic residuosity is hard relative to GenModulus *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\left| \Pr[\mathcal{A}(N, \mathsf{qr}) = 1] - \Pr[\mathcal{A}(N, \mathsf{qnr}) = 1] \right| \leq \mathsf{negl}(n),$$

where in each case the probabilities are taken over the experiment in which $\mathsf{GenModulus}(1^n)$ *is run to give* (N, p, q), qr *is chosen at random from* $\mathcal{QR}_N$, *and* qnr *is chosen at random from* $\mathcal{QNR}_N^{+1}$.

It is crucial in the above that qnr is chosen from $\mathcal{QNR}_N^{+1}$ rather than $\mathcal{QNR}_N$; if qnr were chosen from $\mathcal{QNR}_N$ then with probability $2/3$ it would be the case that $\mathcal{J}_N(x) = -1$ and so distinguishing qnr from a random quadratic residue would be easy. (Recall that $\mathcal{J}_N(x)$ can be computed efficiently even without the factorization of N.)

The quadratic residuosity assumption is simply the assumption that there exists a GenModulus relative to which deciding quadratic residuosity is hard. It is easy to see that if deciding quadratic residuosity is hard relative to GenModulus, then factoring is hard relative to GenModulus as well.

11.1.4 The Goldwasser-Micali Encryption Scheme

The preceding section immediately suggests a public-key encryption scheme for single-bit messages based on the quadratic residuosity assumption:

- The public key is a modulus N, and the secret key is the factorization of N.
- The encryption of the bit '0' is a random quadratic residue, and the encryption of the bit '1' is a random quadratic non-residue with Jacobi symbol $+1$.
- The receiver can decrypt a ciphertext c with its secret key by using the factorization of N to decide whether c is a quadratic residue or not.

Security of this scheme in the sense of Definition 10.3 follows almost trivially from the difficulty of the quadratic residuosity problem as formalized in Definition 11.12.

One thing missing from the above description is a specification of how the sender, who does not know the factorization of N, can choose a random element of $\mathcal{QR}_N$ (in case it wants to encrypt a 0) or a random element of $\mathcal{QNR}_N^{+1}$ (in case it wants to encrypt a 1). The first of these turns out to be easy to do, while the second requires some ingenuity.

Choosing a random quadratic residue. Choosing a random element $y \in \mathcal{QR}_N$ is easy: simply pick a random $x \leftarrow \mathbb{Z}_N^*$ (see Appendix B.2.4) and set $y := x^2 \bmod N$. Clearly $y \in \mathcal{QR}_N$. The fact that y is uniformly distributed in $\mathcal{QR}_N$ follows from the facts that squaring modulo N is a 4-to-1 function (see Section 11.1.2) and that x is chosen at random from $\mathbb{Z}_N^*$. In more detail, fix any $\hat{y} \in \mathcal{QR}_N$ and let us compute the probability that $y = \hat{y}$ after the above procedure. Denote the four square roots of $\hat{y}$ by $\pm\hat{x}, \pm\hat{x}'$. Then:

$$\begin{aligned}\Pr[y = \hat{y}] &= \Pr[x \text{ is a square root of } \hat{y}] \\ &= \Pr\left[x \in \{\pm\hat{x}, \pm\hat{x}'\}\right] \\ &= \frac{4}{|\mathbb{Z}_N^*|} \;=\; \frac{1}{|\mathcal{QR}_N|}\,.\end{aligned}$$

Since the above holds for every $\hat{y} \in \mathcal{QR}_N$, we see that y is distributed uniformly in $\mathcal{QR}_N$.

Choosing a random element of $\mathcal{QNR}_N^{+1}$. In general, it is not known how to choose a random element of $\mathcal{QNR}_N^{+1}$ if the factorization of N is unknown. What saves us in the present context is that *the receiver who generates the keys can help.* Specifically, we modify the scheme as described above so that the receiver additionally chooses a random $z \leftarrow \mathcal{QNR}_N^{+1}$ and includes z as part of its public key. (This is easy for the receiver to do since it knows the factorization of N; see Exercise 11.3.) The sender can choose a random element $y \leftarrow \mathcal{QNR}_N^{+1}$ by choosing a random $x \leftarrow \mathbb{Z}_N^*$ (as above) and setting $y := [z \cdot x^2 \bmod N]$. It follows from Corollary 11.10 that $y \in \mathcal{QNR}_N^{+1}$. We leave it as an exercise to show that y is uniformly distributed in $\mathcal{QNR}_N^{+1}$; we do not use this fact directly in the proof of security given below.

We give a complete description of the Goldwasser-Micali encryption scheme, implementing the above ideas in Construction 11.13.

CONSTRUCTION 11.13

Let GenModulus be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$, and p and q are n-bit primes except with probability negligible in n. Construct a public-key encryption scheme as follows:

- Gen: on input 1^n, run GenModulus(1^n) to obtain (N, p, q), and choose a random $z \leftarrow \mathcal{QNR}_N^{+1}$. The public key is $pk = \langle N, z \rangle$ and the private key is $sk = \langle p, q \rangle$.
- Enc: on input a public key $pk = \langle N, z \rangle$ and a message $m \in \{0, 1\}$, choose a random $x \leftarrow \mathbb{Z}_N^*$ and output the ciphertext
$$c := [z^m \cdot x^2 \bmod N].$$
- Dec: on input a private key $sk = \langle p, q \rangle$ and a ciphertext c, determine whether c is a quadratic residue modulo N using, e.g., Algorithm 11.11. If c is a quadratic residue, output 0; otherwise, output 1.

The Goldwasser-Micali encryption scheme.

THEOREM 11.14 *If the quadratic residuosity problem is hard relative to* GenModulus*, then the Goldwasser-Micali encryption scheme has indistinguishable encryptions under a chosen-plaintext attack.*

PROOF Let Π denote the Goldwasser-Micali encryption scheme. We prove that Π has indistinguishable encryptions in the presence of an eavesdropper; by Theorem 10.10 this implies that it is CPA-secure.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1].$$

Consider the following PPT adversary D that attempts to solve the quadratic residuosity problem relative to $\mathsf{GenModulus}$:

> **Algorithm** D:
> The algorithm is given N and z as input and its goal is to determine if $z \in \mathcal{QR}_N$ or $z \in \mathcal{QNR}^{+1}_N$.
>
> - Set $pk = \langle N, z\rangle$ and run $\mathcal{A}(pk)$ to obtain two single-bit messages m_0, m_1.
> - Choose a random bit b and a random $x \leftarrow \mathbb{Z}^*_N$, and then set $c := [z^{m_b} \cdot x^2 \bmod N]$.
> - Give the ciphertext c to $\mathcal{A}$ and obtain an output bit b'. If $b' = b$, output 1; otherwise, output 0.

Let us analyze the behavior of D. There are two cases to consider:

Case 1: Say the input to D was generated by running $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q), and then choosing a random $z \leftarrow \mathcal{QNR}^{+1}_N$. Then D runs $\mathcal{A}$ on a public key constructed exactly as in Π, and we see that in this case the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to $\mathcal{A}$'s view in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. Since D outputs 1 exactly when the output b' of $\mathcal{A}$ is equal to b, we have that

$$\Pr[D(N, \mathsf{qnr}) = 1] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)\,,$$

where qnr represents a random element of $\mathcal{QNR}^{+1}_N$ as in Definition 11.12.

Case 2: Say the input to D was generated by running $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q), and then choosing a random $z \leftarrow \mathcal{QR}_N$. We claim that the view of $\mathcal{A}$ in this case is *independent* of the bit b. To see this, note that the ciphertext c given to $\mathcal{A}$ is a random quadratic residue regardless of whether a 0 or a 1 is encrypted:

- When a 0 is encrypted, $c = [x^2 \bmod N]$ for a random $x \leftarrow \mathbb{Z}^*_N$ and c is a random quadratic residue.
- When a 1 is encrypted, $c = [z \cdot x^2 \bmod N]$ for a random $x \leftarrow \mathbb{Z}^*_N$. Let $\hat{x} \stackrel{\text{def}}{=} [x^2 \bmod N]$, and note that $\hat{x}$ is a uniformly-distributed element of the group $\mathcal{QR}_N$. Since $z \in \mathcal{QR}_N$, we can apply Lemma 10.20 to conclude that c is uniformly distributed in $\mathcal{QR}_N$ as well.

Since $\mathcal{A}$'s view is independent of b, the probability that $b' = b$ in this case is exactly $\frac{1}{2}$. That is,

$$\Pr[D(N, \mathsf{qr}) = 1] = \frac{1}{2}\,,$$

where qr represents a random element of $\mathcal{QR}_N$ as in Definition 11.12.

Thus,

$$\left| \Pr[D(N, \mathsf{qr}) = 1] - \Pr[D(N, \mathsf{qnr}) = 1] \right| = \left| \varepsilon(n) - \tfrac{1}{2} \right|.$$

By the assumption that the quadratic residuosity problem is hard relative to GenModulus, there exists a negligible function negl such that

$$\left| \varepsilon(n) - \tfrac{1}{2} \right| \leq \mathsf{negl}(n)$$

and so $\varepsilon(n) \leq \frac{1}{2} + \mathsf{negl}(n)$, completing the proof. ■

11.2 The Rabin Encryption Scheme

Security of the Rabin encryption scheme is based on the fact that it is easy to compute square roots modulo a composite number N if the factorization of N is known, yet it appears difficult to compute square roots modulo N when the factorization of N is *unknown*. In fact, as we will see, computing square roots modulo N is *equivalent* to (i.e., it is equally hard as) factoring N. In other words, the factoring assumption implies the difficulty of computing square roots modulo a composite (generated appropriately). Due to this equivalence, a version of the Rabin encryption scheme can be shown to be CPA-secure based solely on the assumption that factoring is hard. This makes the Rabin encryption scheme very attractive, at least from a theoretical point of view. An analogous result is *not* known for RSA encryption, and the RSA problem may potentially be easier than factoring. The same is true of the Goldwasser-Micali encryption scheme, and it may be possible to decide quadratic residuosity modulo N without factoring N.

Interestingly, the Rabin encryption scheme is (superficially, at least) very similar to the RSA encryption scheme yet has the advantage of being based on a potentially weaker assumption. The fact that RSA is more widely-used than the former seems to be due more to historical factors than technical ones; we discuss this further at the end of this section.

11.2.1 Computing Modular Square Roots

The Rabin encryption scheme requires the receiver to compute modular square roots, and so in this section we explore the algorithmic complexity of this problem. We first show an efficient algorithm for computing square roots modulo a prime p, and then extend this algorithm to enable computation of square roots modulo a composite N of *known* factorization. The reader willing to accept the existence of these algorithms on faith can skip to the following

section, where we show that computing square roots modulo a composite N with *unknown* factorization is equivalent to factoring N.

Let p be an odd prime. Computing square roots modulo p is relatively simple when $p = 3 \bmod 4$, and much more involved when $p = 1 \bmod 4$. (The easier case is all we need for the Rabin encryption scheme as presented in Section 11.2.3; we include the second case for completeness.) In both cases, we show how to compute one of the square roots of a quadratic residue $a \in \mathbb{Z}_p^*$. Note that if x is one of the square roots of a, then $[-x \bmod p]$ is the other.

We tackle the easier case first. Say $p = 3 \bmod 4$, meaning we can write $p = 4i + 3$ for some integer i. Since $a \in \mathbb{Z}_p^*$ is a quadratic residue, we have $\mathcal{J}_p(a) = 1 = a^{\frac{p-1}{2}} \bmod p$ (see Proposition 11.2). Multiplying both sides by a we obtain

$$a = a^{\frac{p-1}{2}+1} = a^{2i+2} = \left(a^{i+1}\right)^2 \bmod p\,,$$

and so $a^{i+1} = a^{\frac{p+1}{4}} \bmod p$ is a square root of a. That is, we can compute a square roots of a modulo p as $x := [a^{\frac{p+1}{4}} \bmod p\,]$.

It is crucial above that $(p+1)/2$ is *even* because this ensures that $(p+1)/4$ is an *integer* (this is necessary in order for $a^{\frac{p+1}{4}} \bmod p$ to be well-defined; recall that the exponent must be an integer). This approach does not succeed when $p = 1 \bmod 4$, in which case $p+1$ is an integer that is *not* divisible by 4.

When $p = 1 \bmod 4$ we proceed slightly differently. Motivated by the above approach, we might think to search for an *odd* integer r for which it holds that $a^r = 1 \bmod p$. Then, as above, $a^{r+1} = a \bmod p$ and $a^{\frac{r+1}{2}} \bmod p$ would be a square root of a with $(r+1)/2$ an integer. Though we will not be able to do this, we *can* do something just as good: we will find an odd integer r along with an element $b \in \mathbb{Z}_p^*$ and an *even* integer r' such that

$$a^r \cdot b^{r'} = 1 \bmod p.$$

Then $a^{r+1} \cdot b^{r'} = a \bmod p$ and $a^{\frac{r+1}{2}} \cdot b^{\frac{r'}{2}} \bmod p$ is a square root of a (with the exponents $(r+1)/2$ and $r'/2$ being integers).

Example 11.15

Take $p = 29$ and $a = 22$. Then $22^7 \cdot 2^{14} = 1 \bmod 29$, and so we have that $15 = 22^{(7+1)/2} \cdot 2^{14/2} = 22^4 \cdot 2^7 \bmod 29$ is a square root of 22 modulo 29. The other square root is, of course, $-15 = 14 \bmod 29$. ♢

We now describe the general approach to finding r, b, and r' with the stated properties. Let $\frac{p-1}{2} = 2^\ell \cdot m$ where ℓ, m are integers with $\ell \geq 1$ and m odd.[2] Since a is a quadratic residue, we know that

$$a^{2^\ell m} = a^{\frac{p-1}{2}} = 1 \bmod p. \tag{11.3}$$

[2]The integers ℓ and m can be computed easily by taking out factors of 2 from $(p-1)/2$.

This means that $a^{2^{\ell}m/2} = a^{2^{\ell-1}m} \bmod p$ is a square root of 1. The square roots of 1 modulo p are $\pm 1 \bmod p$, so we know that $a^{2^{\ell-1}m} = \pm 1 \bmod p$. If $a^{2^{\ell-1}m} = 1 \bmod p$, we are in the same situation as in Equation (11.3) except that the exponent of a is now divisible by a smaller power of 2. This is progress in the right direction: if we can get to the point where the exponent of a is not divisible by *any* power of 2 (as would be the case here if $\ell = 1$), then the exponent of a is *odd* and we can compute a square root as discussed earlier. We give an example, and discuss in a moment how to deal with the case when $a^{2^{\ell-1}m} = -1 \bmod p$.

Example 11.16
Take $p = 29$ and $a = 7$. Since 7 is a quadratic residue modulo 29, we have $7^{14} \bmod 29 = 1$ and we know that $7^7 \bmod 29$ is a square root of 1. In fact,

$$7^7 = 1 \bmod 29,$$

and the exponent 7 is odd. So $7^{(7+1)/2} = 7^4 = 23 \bmod 29$ is a square root of 7 modulo 29. ♢

To summarize where things stand: we begin with $a^{2^{\ell}m} = 1 \bmod p$ and we pull factors of 2 out of the exponent of a until one of two things happen: either $a^m = 1 \bmod p$, or $a^{2^{\ell'}m} = -1 \bmod p$ for some $\ell' < \ell$. In the first case, since m is odd we can immediately compute a square root of a as in Example 11.16. In the second case, we will "restore" the $+1$ on the right-hand side of the equation by multiplying each side of the equation by $-1 \bmod p$. However, as motivated at the beginning of this discussion, we want to achieve this by multiplying the left-hand side of the equation by some element b raised to an *even* power. If we have available a quadratic *non*-residue $b \in \mathbb{Z}_p^*$, this is easy: since $b^{2^{\ell}m} = b^{\frac{p-1}{2}} = -1 \bmod p$ we have

$$a^{2^{\ell'}m} \cdot b^{2^{\ell}m} = (-1)(-1) = +1 \bmod p.$$

We can now proceed as before, taking a square root of the entire left-hand side to reduce the largest power of 2 dividing the exponent of a, and multiplying by $b^{2^{\ell}m}$ (as needed) so the right-hand side is always $+1$. Observe that the exponent of b is always divisible by a larger power of 2 than the exponent of a (and so we can indeed take square roots by dividing by 2 in both exponents). We continue performing these steps until the exponent of a is odd, and can then compute a square root of a as described earlier. Pseudocode for this algorithm, which gives another way of viewing what is going on, is given below in Algorithm 11.17. It can be verified that the algorithm runs in polynomial time given a quadratic non-residue b.

One point we have not yet addressed is how to find b in the first place. Actually, no *deterministic* polynomial-time algorithm for finding a quadratic

ALGORITHM 11.17
Computing square roots modulo a prime

Input: Prime p; quadratic residue $a \in \mathbb{Z}_p^*$
Output: A square root of a

case $p = 3 \bmod 4$:
  return $[a^{\frac{p+1}{4}} \bmod p]$
case $p = 1 \bmod 4$:
  let b be a quadratic non-residue modulo p
  compute ℓ and m odd with $2^\ell \cdot m = \frac{p-1}{2}$
  $r := 2^\ell \cdot m,\ r' := 0$
  for $i = \ell$ to 1 {
  /* maintain the invariant $a^r \cdot b^{r'} = 1 \bmod p$ */
    $r := r/2,\ r' := r'/2$
    if $a^r \cdot b^{r'} = -1 \bmod p$
      $r' := r' + 2^\ell \cdot m$
  }
  /* now $r = m,\ r'$ is even, and $a^r \cdot b^{r'} = 1 \bmod p$ */
  return $\left[a^{\frac{r+1}{2}} \cdot b^{\frac{r'}{2}} \bmod p\right]$

non-residue modulo p is known. Fortunately, it is easy to find a quadratic non-residue probabilistically: simply choose random elements of $\mathbb{Z}_p^*$ until a quadratic non-residue is found. This works because exactly half the elements of $\mathbb{Z}_p^*$ are quadratic non-residues, and because a polynomial-time algorithm for deciding quadratic residuosity modulo a prime is known (see Section 11.1.1 for proofs of both these statements). This means that the algorithm we have shown is actually randomized when $p = 1 \bmod 4$; a deterministic polynomial-time algorithm for computing square roots in this case is not known.

Example 11.18
Here we consider the "worst case," when taking a square root always gives -1. Let $a \in \mathbb{Z}_p^*$ be the element whose square root we are trying to compute; let $b \in \mathbb{Z}_p^*$ be a quadratic non-residue; and let $\frac{p-1}{2} = 2^3 \cdot m$ where m is odd.

In the first step, we have $a^{2^3 m} = 1 \bmod p$. Since $a^{2^3 m} = \left(a^{2^2 m}\right)^2$ and the square roots of 1 are ± 1, this means that $a^{2^2 m} = \pm 1 \bmod p$; assuming the worst case, $a^{2^2 m} = -1 \bmod p$. So, we multiply by $b^{\frac{p-1}{2}} = b^{2^3 m} = -1 \bmod p$ to obtain

$$a^{2^2 m} \cdot b^{2^3 m} = 1 \bmod p.$$

In the second step, we observe that $a^{2m} \cdot b^{2^2 m}$ is a square root of 1; again assuming the worst case, we thus have $a^{2m} \cdot b^{2^2 m} = -1 \bmod p$. Multiplying by $b^{2^3 m}$ to "correct" this gives

$$a^{2m} \cdot b^{2^2 m} \cdot b^{2^3 m} = 1 \bmod p.$$

In the third step, taking square roots and assuming the worst case (as above) we obtain $a^m \cdot b^{2m} \cdot b^{2^2 m} = -1 \bmod p$; multiplying by the "correction factor" $b^{2^3 m}$ we get

$$a^m \cdot b^{2m} \cdot b^{2^2 m} \cdot b^{2^3 m} = 1 \bmod p.$$

We are now where we want to be. To conclude the algorithm, multiply both sides by a to obtain

$$a^{m+1} \cdot b^{2m+2^2 m+2^3 m} = a \bmod p.$$

Since m is odd, $(m+1)/2$ is an integer and $a^{\frac{m+1}{2}} \cdot b^{m+2m+2^2 m} \bmod p$ is a square root of a. ♢

Example 11.19

Here we work out a concrete example. Let $p = 17$, $a = 4$, and $b = 3$. Note that here $(p-1)/2 = 2^3$ and $m = 1$.

We begin with $4^{2^3} = 1 \bmod 17$. So 4^{2^2} should be equal to $\pm 1 \bmod 17$; by calculation, we see that $4^{2^2} = 1 \bmod 17$ and so no correction term is needed in this step.

Continuing, we know that 4^2 is a square root of 1 and so must be equal to $\pm 1 \bmod 17$; calculation gives $4^2 = -1 \bmod 17$. Multiplying by 3^{2^3} gives $4^2 \cdot 3^{2^3} = 1 \bmod 17$.

Finally, we consider $4 \cdot 3^{2^2} = 1 \bmod 17$. We are now almost done: multiplying both sides by 4 gives $4^2 \cdot 3^{2^2} = 4 \bmod 17$ and so $4 \cdot 3^2 = 2 \bmod 17$ is a square root of 4. ♢

Computing Square Roots Modulo N

It is not hard to see that the algorithm we have shown for computing square roots modulo a prime can be extended easily to the case of computing square roots modulo a composite $N = pq$ of known factorization. Specifically, let $a \in \mathbb{Z}_N^*$ be a quadratic residue with $a \leftrightarrow (a_p, a_q)$ via the Chinese remainder theorem. Computing the square roots x_p, x_q of a_p, a_q modulo p and q, respectively, gives a square root (x_p, x_q) of a (see Section 11.1.2). Given x_p and x_q, the representation x corresponding to (x_p, x_q) can be recovered as discussed in Section 7.1.5. Writing out these steps explicitly: to compute a square root of a modulo an integer $N = pq$ of known factorization, do:

- Compute $a_p := [a \bmod p]$ and $a_q := [a \bmod q]$.
- Using Algorithm 11.17, compute a square root x_p of a_p modulo p and a square root x_q of a_q modulo q

- Convert from the representation $(x_p, x_q) \in \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ to $x \in \mathbb{Z}_N^*$ with $x \leftrightarrow (x_p, x_q)$. Output x, which is a square root of a modulo N.

It is easy to modify the algorithm so that it returns all four square roots of a.

11.2.2 A Trapdoor Permutation Based on Factoring

We have seen that computing square roots modulo N can be carried out in polynomial time if the factorization of N is known. We show here that, in contrast, computing square roots modulo a composite N of *unknown* factorization is as hard as factoring N.

More formally, let GenModulus be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$ and p and q are n-bit primes except with probability negligible in n. Consider the following experiment for a given algorithm $\mathcal{A}$ and parameter n:

The square root computation experiment $\mathsf{SQR}_{\mathcal{A},\mathsf{GenModulus}}(n)$:

1. *Run* $\mathsf{GenModulus}(1^n)$ *to obtain output* N, p, q.
2. *Choose* $y \leftarrow \mathcal{QR}_N$.
3. $\mathcal{A}$ *is given* (N, y), *and outputs* $x \in \mathbb{Z}_N^*$.
4. *The output of the experiment is defined to be 1 if* $x^2 = y \bmod N$, *and 0 otherwise.*

DEFINITION 11.20 *We say that* computing square roots is hard relative to GenModulus *if for all probabilistic polynomial-time algorithms* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{SQR}_{\mathcal{A},\mathsf{GenModulus}}(n) = 1] \leq \mathsf{negl}(n).$$

It is easy to see that if computing square roots is hard relative to GenModulus then factoring must be hard relative to GenModulus too: if moduli N output by GenModulus could be factored easily then it would be easy to compute square roots modulo N by first factoring N and then applying the algorithm discussed in the previous section. Our aim now is to show the converse: that if factoring is *hard* relative to GenModulus then so is the problem of computing square roots. We emphasize again that such a result is not known for the RSA problem or the problem of deciding quadratic residuosity.

The key is the following lemma, which says that two "unrelated" square roots of any element in $\mathbb{Z}_N^*$ can be used to factor N.

LEMMA 11.21 *Let* $N = pq$ *with* p, q *distinct, odd primes. Given* $x, \hat{x}$ *such that* $x^2 = y = \hat{x}^2 \bmod N$ *but* $x \neq \pm\hat{x} \bmod N$, *it is possible to factor* N *in time polynomial in* $\|N\|$.

PROOF We claim that either $\gcd(N, x + \hat{x})$ or $\gcd(N, x - \hat{x})$ is equal to one of the prime factors of N.[3] Since gcd computations can be carried out in polynomial time (see Appendix B.1.2), this proves the lemma.

If $x^2 = \hat{x}^2 \bmod N$ then

$$0 = x^2 - \hat{x}^2 = (x - \hat{x}) \cdot (x + \hat{x}) \bmod N,$$

and so $N \mid (x - \hat{x})(x + \hat{x})$. Then $p \mid (x - \hat{x})(x + \hat{x})$ and so p divides one of these terms. Say $p \mid (x + \hat{x})$ (the proof proceeds similarly if $p \mid (x - \hat{x})$). If $q \mid (x + \hat{x})$ then $N \mid (x + \hat{x})$, but this cannot be the case since $x \neq -\hat{x} \bmod N$. So $q \not| x + \hat{x}$ and $\gcd(N, x + \hat{x}) = p$. ■

An alternative way of proving the above is to look at what happens in the Chinese remaindering representation. Say $x \leftrightarrow (x_p, x_q)$. Then, because x and $\hat{x}$ are square roots of the same value y, we know that $\hat{x}$ corresponds to either $(-x_p, x_q)$ or $(x_p, -x_q)$. (It cannot correspond to (x_p, x_q) or $(-x_p, -x_q)$ since the first corresponds to x while the second corresponds to $[-x \bmod N]$, and both possibilities are ruled out by the assumption of the lemma.) Say $\hat{x} \leftrightarrow (-x_p, x_q)$. Then

$$[x + \hat{x} \bmod N] \leftrightarrow (x_p, x_q) + (-x_p, x_q) = (0, [2x_q \bmod q]),$$

and we see that $x + \hat{x} = 0 \bmod p$ while $x + \hat{x} \neq 0 \bmod q$. It follows that $\gcd(N, x + \hat{x}) = p$, a factor of N.

We can now prove the main result of this section.

THEOREM 11.22 *If factoring is hard relative to* GenModulus*, then computing square roots is hard relative to* GenModulus.

PROOF Let $\mathcal{A}$ be a probabilistic polynomial-time algorithm, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr\left[\mathsf{SQR}_{\mathcal{A},\mathsf{GenModulus}}(n) = 1\right].$$

Consider the following probabilistic polynomial-time algorithm $\mathcal{A}_{\text{fact}}$ that attempts to factor moduli output by GenModulus:

Algorithm $\mathcal{A}_{\text{fact}}$:
The algorithm is given a modulus N as input.

- Choose random $x \leftarrow \mathbb{Z}_N^*$ and compute $y := [x^2 \bmod N]$.
- Run $\mathcal{A}(N, y)$ to obtain output $\hat{x}$.

[3] In fact, both of these are equal to one of the prime factors of N but it is easier to prove what we have claimed and this is anyway sufficient.

- If $\hat{x}^2 = y \bmod N$ and $\hat{x} \neq \pm x \bmod N$, then factor N using Lemma 11.21.

By Lemma 11.21, we know that $\mathcal{A}_{\text{fact}}$ succeeds in factoring N exactly when $\hat{x} \neq \pm x \bmod N$ and $\hat{x}^2 = y \bmod N$. That is,

$$\begin{aligned}
&\Pr[\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenModulus}}(n) = 1] \\
&= \Pr\left[\hat{x} \neq \pm x \bmod N \wedge \hat{x}^2 = y \bmod N\right] \\
&= \Pr\left[\hat{x} \neq \pm x \bmod N \,\middle|\, \hat{x}^2 = y \bmod N\right] \cdot \Pr\left[\hat{x}^2 = y \bmod N\right], \quad (11.4)
\end{aligned}$$

where the above probabilities all refer to experiment $\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenModulus}}(n)$ (refer to Section 7.2.3 for a description of this experiment). In the experiment, the modulus N given as input to $\mathcal{A}_{\text{fact}}$ is generated by $\mathsf{GenModulus}(1^n)$, and y is a random quadratic residue modulo N since x was chosen uniformly at random from $\mathbb{Z}_N^*$ (see Section 11.1.4). So the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}_{\text{fact}}$ is distributed exactly as $\mathcal{A}$'s view in experiment $\mathsf{SQR}_{\mathcal{A},\mathsf{GenModulus}}(n)$. Therefore,

$$\Pr\left[\hat{x}^2 = y \bmod N\right] = \Pr\left[\mathsf{SQR}_{\mathcal{A},\mathsf{GenModulus}}(n) = 1\right] = \varepsilon(n). \quad (11.5)$$

Conditioned on the value of the quadratic residue y used in experiment $\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenModulus}}(n)$, the value x is equally likely to be any of the four possible square roots of y. This means that from the point of view of algorithm $\mathcal{A}$ (being run as a subroutine by $\mathcal{A}_{\text{fact}}$), x is equally likely to be each of the four square roots of y. This in turn means that, conditioned on $\mathcal{A}$ outputting *some* square root $\hat{x}$ of y, the probability that $\hat{x} = \pm x \bmod N$ is exactly $1/2$. (We stress that we do not make any assumption about how $\hat{x}$ is distributed among the square roots of y, and in particular are not assuming here that $\mathcal{A}$ outputs a random square root of y. Rather we are using the fact that x is uniformly distributed among the square roots of y.) That is,

$$\Pr\left[\hat{x} \neq \pm x \bmod N \,\middle|\, \hat{x}^2 = y \bmod N\right] = \frac{1}{2}. \quad (11.6)$$

Combining Equations (11.4)–(11.6), we see that

$$\Pr[\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenModulus}}(n) = 1] = \frac{1}{2} \cdot \varepsilon(n).$$

Since factoring is hard relative to $\mathsf{GenModulus}$, there exists a negligible function negl such that

$$\Pr[\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenModulus}}(n) = 1] \leq \mathsf{negl}(n).$$

This implies $\varepsilon(n)/2 \leq \mathsf{negl}(n)$ or, equivalently, $\varepsilon(n) \leq 2 \cdot \mathsf{negl}(n)$, completing the proof. ∎

The previous theorem leads directly to a family of one-way functions (see Definition 7.70) based on any $\mathsf{GenModulus}$ relative to which factoring is hard:

- Algorithm Gen, on input 1^n, runs $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q) and outputs $I = N$. The domain $\mathcal{D}_I$ is $\mathbb{Z}_N^*$ and the range $\mathcal{R}_I$ is $\mathcal{QR}_N$.
- Algorithm Samp, on input N, chooses a random element $x \leftarrow \mathbb{Z}_N^*$.
- Algorithm f, on input N and $x \in \mathbb{Z}_N^*$, outputs $[x^2 \bmod N]$.

The fact that this family is one-way follows from the assumption that factoring is hard and from the fact that inverting f is equivalent to factoring.

We can turn this into a family of one-way *permutations* by using moduli N of a special form and letting $\mathcal{D}_I$ be a subset of $\mathbb{Z}_N^*$. (See the exercises for other way to make this a permutation.) Say $N = pq$ is a *Blum integer* if p and q are distinct primes with $p = q = 3 \bmod 4$. The key to building a permutation is the following proposition.

PROPOSITION 11.23 *Let N be a Blum integer. Then every quadratic residue modulo N has exactly one square root that is also a quadratic residue.*

PROOF Say $N = pq$ with $p = q = 3 \bmod 4$. Using Proposition 11.2, we see that -1 is not a quadratic residue modulo p or q. This is because for $p = 3 \bmod 4$ it holds that $p = 4i + 3$ for some i and so

$$(-1)^{\frac{p-1}{2}} = (-1)^{2i+1} = -1 \bmod p$$

(because $2i+1$ is odd). Now let $y \leftrightarrow (y_p, y_q)$ be an arbitrary quadratic residue modulo N with the four square roots

$$(x_p,\, x_q), \quad (-x_p,\, x_q), \quad (x_p,\, -x_q), \quad (-x_p,\, -x_q).$$

We claim that exactly one of these is a quadratic residue modulo N. To see this, assume $\mathcal{J}_p(x_p) = +1$ and $\mathcal{J}_q(x_q) = -1$ (the proof proceeds similarly in any other case). Using Proposition 11.4, we have

$$\mathcal{J}_q(-x_q) = \mathcal{J}_q(-1) \cdot \mathcal{J}_q(x_q) = +1\,,$$

and so $(x_p, -x_q)$ corresponds to a quadratic residue modulo N (using Proposition 11.6). Similarly, $\mathcal{J}_p(-x_p) = -1$ and so none of the other square roots of y are quadratic residues modulo N. ■

Expressed differently, the above proposition says that when N is a Blum integer, the function $f_N : \mathcal{QR}_N \to \mathcal{QR}_N$ given by $f_N(x) = [x^2 \bmod N]$ is a permutation over $\mathcal{QR}_N$. Modifying the sampling algorithm Samp, above, to choose a random $x \leftarrow \mathcal{QR}_N$ (which, as we have already seen, can be done easily by choosing random $r \leftarrow \mathbb{Z}_N^*$ and setting $x := [r^2 \bmod N]$) gives a family of one-way *permutations*. Finally, because square roots modulo N can be computed in polynomial time given the factorization of N, a straightforward modification yields a family of *trapdoor* permutations based on any

GenModulus relative to which factoring is hard. This is sometimes called the *Rabin* family of trapdoor permutations. In summary:

THEOREM 11.24 *Let* GenModulus *be an algorithm that, on input* 1^n*, outputs* (N, p, q) *where* $N = pq$ *and* p *and* q *are distinct primes (except possibly with negligible probability) with* $p = q = 3 \bmod 4$*. If factoring is hard relative to* GenModulus*, then there exists a family of trapdoor permutations.*

11.2.3 The Rabin Encryption Scheme

The Rabin trapdoor permutation suggests a "textbook Rabin" encryption scheme by analogy with "textbook RSA" encryption. Such a scheme is deterministic and therefore is not CPA-secure. A better approach is to rely on the results of Section 10.7.2, where we showed that any trapdoor permutation can be used to construct a CPA-secure public-key encryption scheme. To apply the transformation shown there to the Rabin family of trapdoor permutations introduced in the previous section, we need a predicate that is hard-core for this family (see Definition 10.26). It can be shown that the least-significant bit $\mathsf{lsb}(\cdot)$ constitutes a hard-core predicate. That is, let GenModulus be an algorithm outputting Blum integers relative to which factoring is hard. Then for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function negl such that

$$\Pr[\mathcal{A}(N, [x^2 \bmod N]) = \mathsf{lsb}(x)] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over the experiment in which $\mathsf{GenModulus}(1^n)$ outputs (N, p, q) and then x is chosen at random from $\mathcal{QR}_N$. Plugging this into Construction 10.27 gives a concrete example of a public-key encryption scheme whose security can be based on the factoring assumption. We describe this formally in Construction 11.25.

We do not prove the following theorem here; see Exercise 10.10 of Chapter 10 (and the reference there) for an idea as to how a proof would proceed.

THEOREM 11.26 *If factoring is hard relative to* GenModulus*, then Construction 11.25 has indistinguishable encryptions under a chosen-plaintext attack.*

One could also consider a "padded Rabin" encryption scheme, constructed by analogy with the padded RSA encryption scheme of Section 10.4.3, that is more efficient than Construction 11.25. In this case, however, provable security based on factoring is not known to hold. See Exercise 11.14 for one example of how this could be done.

CONSTRUCTION 11.25

Let GenModulus be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$ and p and q are n-bit primes (except with probability negligible in n) with $p = q = 3 \bmod 4$. Construct a public-key encryption scheme as follows:

- Gen: on input 1^n run GenModulus(1^n) to obtain (N, p, q). The public key is N, and the private key is $\langle p, q \rangle$.
- Enc: on input a public-key N and message $m \in \{0, 1\}$, choose a random $x \leftarrow \mathcal{QR}_N$ and output the ciphertext

$$c := \langle [x^2 \bmod N], \mathsf{lsb}(x) \oplus m \rangle.$$

- Dec: on input a private key $\langle p, q \rangle$ and a ciphertext $\langle c, c' \rangle$, compute the unique $x \in \mathcal{QR}_N$ such that $x^2 = c \bmod N$, and output $\mathsf{lsb}(x) \oplus c'$.

The Rabin encryption scheme.

Rabin Encryption vs. RSA Encryption

It is worthwhile to remark on the similarities and differences between the Rabin and RSA cryptosystems. (The discussion here applies to any cryptographic construction — not necessarily a public-key encryption scheme — based on the Rabin or RSA trapdoor permutations.)

At a basic level, the RSA and Rabin trapdoor permutations appear quite similar, with squaring in the case of Rabin corresponding to taking $e = 2$ in the case of RSA. (Of course, '2' is *not* relatively prime to $\phi(N)$ and so Rabin is not a special case of RSA.) In terms of the security offered by each construction, we have noted that hardness of computing modular square roots is equivalent to hardness of factoring, while hardness of solving the RSA problem is not known to be implied by the hardness of factoring. The Rabin trapdoor permutation is thus based on a potentially *weaker* assumption. It is theoretically possible that someone might develop an efficient algorithm for solving the RSA problem, yet computing square roots will remain hard. More plausible is someone will propose an algorithm that solves the RSA problem in less time than it takes to factor (but that still requires super-polynomial time). Lemma 11.21 ensures, however, that computing square roots modulo N can never be much faster than the best available algorithm for factoring N.

In terms of their efficiency, the RSA and Rabin permutations are essentially the same. Actually, if a large exponent e is used in the case of RSA then computing eth powers (as in RSA) is slightly slower than squaring (as in Rabin). On the other hand, a bit more care is required when working with the Rabin permutation since it is only a permutation over a *subset* of $\mathbb{Z}_N^*$, in contrast to RSA which gives a permutation over all of $\mathbb{Z}_N^*$.

A "textbook Rabin" encryption scheme, constructed in a manner exactly analogous to textbook RSA encryption, is vulnerable to a chosen-ciphertext

attack that enables an adversary to learn the entire private key of the receiver (see Exercise 11.12). Although textbook RSA is vulnerable to a chosen-ciphertext attack that recovers the entire message, there is no known chosen-ciphertext attack on textbook RSA that recovers the entire private key. It is possible that the existence of such an attack on "textbook Rabin" influenced cryptographers, early on, to reject the use of Rabin encryption entirely.

In summary, the RSA permutation is much more widely used in practice than the Rabin permutation, but in light of the above this appears to be due more to historical accident than to any compelling technical justification.

11.3 The Paillier Encryption Scheme

The Paillier encryption scheme, like the RSA, Goldwasser-Micali, and Rabin encryption schemes, is based on the hardness of factoring a composite number N that is the product of two primes. (We emphasize that, with the exception of Rabin encryption, security of these schemes is not known to be *equivalent to* the hardness of factoring.) The Paillier encryption scheme is more efficient than the Goldwasser-Micali cryptosystem, as well as the provably-secure RSA and Rabin schemes of Theorems 10.19 and 11.26, respectively. Perhaps more importantly, the Paillier encryption scheme possesses some nice *homomorphic* properties we will discuss further in Section 11.3.3. We remark, however, that it relies on a newer and less studied hardness assumption.

The Paillier encryption scheme utilizes the group $\mathbb{Z}_{N^2}^*$. A useful characterization of this group is given by the following proposition which says, among other things, that $\mathbb{Z}_{N^2}^*$ is isomorphic to $\mathbb{Z}_N \times \mathbb{Z}_N^*$ (see Definition 7.23) for N of the form we will be interested in.[4] We will prove the proposition in the next section.

PROPOSITION 11.27 *Let $N = pq$, where p, q are distinct odd primes of the same length. Then:*

1. $\gcd(N, \phi(N)) = 1$.
2. *For any integer $a \geq 0$, we have $(1+N)^a = (1+aN) \bmod N^2$.*

   As a consequence, the order of $(1+N)$ in $\mathbb{Z}_{N^2}^*$ is N. That is, $(1+N)^N = 1 \bmod N^2$ and $(1+N)^a \neq 1 \bmod N^2$ for any $1 \leq a < N$.
3. *$\mathbb{Z}_N \times \mathbb{Z}_N^*$ is isomorphic to $\mathbb{Z}_{N^2}^*$, with isomorphism $f\colon \mathbb{Z}_N \times \mathbb{Z}_N^* \to \mathbb{Z}_{N^2}^*$ given by*

$$f(a, b) = \left[(1+N)^a \cdot b^N \bmod N^2\right].$$

[4] Recall that $\mathbb{Z}_N$ is a group under *addition* modulo N, while $\mathbb{Z}_N^*$ is a group under *multiplication* modulo N.

In light of the final claim of the above proposition, we introduce some convenient shorthand. With N understood, and $x \in \mathbb{Z}_{N^2}^*$, $a \in \mathbb{Z}_N$, $b \in \mathbb{Z}_N^*$, we write $x \leftrightarrow (a, b)$ if $f(a, b) = x$ where f is the isomorphism from the proposition above. One way to think about this notation is that it means "x in $\mathbb{Z}_{N^2}^*$ *corresponds to* (a, b) in $\mathbb{Z}_N \times \mathbb{Z}_N^*$". We have used the same notation throughout this book in the context of the isomorphism $\mathbb{Z}_N^* \simeq \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ given by the Chinese remainder theorem; we keep the notation because in both cases it refers to an isomorphism of groups. Nevertheless, there should be no confusion since the group $\mathbb{Z}_{N^2}^*$ and the above proposition are only used in this section (and the Chinese remainder theorem is not used in this section). We remark that here the isomorphism — but not its inverse — is efficiently computable even without the factorization of N.

Section 11.3.1 is dedicated to a proof of Proposition 11.27. The reader who is willing to accept the proposition on faith can skip directly to Section 11.3.2.

11.3.1 The Structure of $\mathbb{Z}_{N^2}^*$

In this section, we prove Proposition 11.27 claim-by-claim. Throughout, we let N, p, q be as in the proposition.

CLAIM 11.28 *For N, p, q as in Proposition 11.27,* $\gcd(N, \phi(N)) = 1$.

PROOF Recall that $\phi(N) = (p-1)(q-1)$. Assume $p > q$ without loss of generality. Since p is prime and $p > p - 1 > q - 1$, clearly $\gcd(p, \phi(N)) = 1$. Similarly, $\gcd(q, q-1) = 1$. Now, if $\gcd(q, p-1) \neq 1$ then $\gcd(q, p-1) = q$ since q is prime. But then $(p-1)/q \geq 2$, contradicting the assumption that p and q have the same length. ■

CLAIM 11.29 *For $a \geq 0$ an integer, we have $(1+N)^a = 1 + aN \bmod N^2$. Thus, the order of $(1+N)$ in $\mathbb{Z}_{N^2}^*$ is N.*

PROOF Using the binomial expansion theorem (Theorem A.1):

$$(1+N)^a = \sum_{i=0}^{a} \binom{a}{i} N^i.$$

Reducing the right-hand side modulo N^2, all terms with $i \geq 2$ become 0 and so $(1+N)^a = 1 + aN \bmod N^2$. The smallest non-zero a such that $(1+N)^a = 1 \bmod N^2$ is therefore $a = N$. ■

CLAIM 11.30 *The group $\mathbb{Z}_N \times \mathbb{Z}_N^*$ is isomorphic to the group $\mathbb{Z}_{N^2}^*$, with isomorphism $f\colon \mathbb{Z}_N \times \mathbb{Z}_N^* \to \mathbb{Z}_{N^2}^*$ given by $f(a, b) = [(1+N)^a \cdot b^N \bmod N^2]$.*

PROOF Note that $(1+N)^a \cdot b^N$ does not have a factor in common with N^2 since $\gcd((1+N), N^2) = 1$ and $\gcd(b, N^2) = 1$ (because $b \in \mathbb{Z}_N^*$). So $[(1+N)^a \cdot b^N \bmod N^2]$ lies in $\mathbb{Z}_{N^2}^*$. We now prove that f is an isomorphism.

We first show that f is a bijection. Since

$$\begin{aligned}|\mathbb{Z}_{N^2}^*| = \phi(N^2) = p \cdot (p-1) \cdot q \cdot (q-1) &= pq \cdot (p-1)(q-1) \\ &= |\mathbb{Z}_N| \cdot |\mathbb{Z}_N^*| = |\mathbb{Z}_N \times \mathbb{Z}_N^*|\end{aligned}$$

(see Theorem 7.19 for the second equality), it suffices to show that f is one-to-one. Say $a_1, a_2 \in \mathbb{Z}_N$ and $b_1, b_2 \in \mathbb{Z}_N^*$ are such that $f(a_1, b_1) = f(a_2, b_2)$. Then:

$$(1+N)^{a_1 - a_2} \cdot (b_1/b_2)^N = 1 \bmod N^2. \tag{11.7}$$

(Note that $b_2 \in \mathbb{Z}_N^*$ and thus $b_2 \in \mathbb{Z}_{N^2}^*$, and so b_2 has a multiplicative inverse modulo N^2.) Raising both sides to the power $\phi(N)$ and using the fact that the order of $\mathbb{Z}_{N^2}^*$ is $\phi(N^2) = N \cdot \phi(N)$ we obtain

$$\begin{aligned}(1+N)^{(a_1-a_2)\cdot\phi(N)} \cdot (b_1/b_2)^{N\cdot\phi(N)} &= 1 \bmod N^2 \\ \Rightarrow (1+N)^{(a_1-a_2)\cdot\phi(N)} &= 1 \bmod N^2 .\end{aligned}$$

By Claim 11.29, $(1+N)$ has order N modulo N^2. Applying Proposition 7.50, we see that $(a_1 - a_2) \cdot \phi(N) = 0 \bmod N$ and so N divides $(a_1 - a_2) \cdot \phi(N)$. Since $\gcd(N, \phi(N)) = 1$ by Claim 11.28, it follows that $N \mid (a_1 - a_2)$. Since $a_1, a_2 \in \mathbb{Z}_N$, this can only occur if $a_1 = a_2$.

Returning to Equation (11.7) and setting $a_1 = a_2$, we thus have $b_1^N = b_2^N \bmod N^2$. This implies $b_1^N = b_2^N \bmod N$. Since N is relatively prime to $\phi(N)$, the order of $\mathbb{Z}_N^*$, exponentiation to the power N is a bijection in $\mathbb{Z}_N^*$ (cf. Corollary 7.17). This means that $b_1 = b_2 \bmod N$; since $b_1, b_2 \in \mathbb{Z}_N^*$, we have $b_1 = b_2$. We conclude that f is one-to-one, and hence a bijection.

To show that f is an isomorphism, we show that $f(a_1, b_1) \cdot f(a_2, b_2) = f(a_1 + a_2, b_1 \cdot b_2)$. (Note that multiplication on the left-hand side of the equality takes place modulo N^2, while addition/multiplication on the right-hand side takes place modulo N.) We have:

$$\begin{aligned}f(a_1, b_1) \cdot f(a_2, b_2) &= \left((1+N)^{a_1} \cdot b_1^N\right) \cdot \left((1+N)^{a_2} \cdot b_2^N\right) \bmod N^2 \\ &= (1+N)^{a_1+a_2} \cdot (b_1 b_2)^N \bmod N^2.\end{aligned}$$

Since $(1+N)$ has order N modulo N^2 (by Claim 11.29), we can apply Proposition 7.49 and obtain

$$\begin{aligned}f(a_1, b_1) \cdot f(a_2, b_2) &= (1+N)^{a_1+a_2} \cdot (b_1 b_2)^N \bmod N^2 \\ &= (1+N)^{a_1+a_2 \bmod N} \cdot (b_1 b_2)^N \bmod N^2. \end{aligned} \tag{11.8}$$

We are not yet done, since $b_1 b_2$ in Equation (11.8) represents multiplication modulo N^2 whereas we would like it to be modulo N. Let $b_1 b_2 = r + \gamma N$,

where γ, r are integers with $1 \leq r < N$ (r cannot be 0 since $b_1, b_2 \in \mathbb{Z}_N^*$ and so their product cannot be divisible by N). Note that $r = b_1 b_2 \bmod N$. We also have

$$\begin{aligned}(b_1 b_2)^N &= (r + \gamma N)^N \bmod N^2 \\ &= \sum_{k=0}^{N} \binom{N}{k} r^{N-k} (\gamma N)^k \bmod N^2 \\ &= r^N + N \cdot r^{N-1} \cdot (\gamma N) = r^N = (b_1 b_2 \bmod N)^N \bmod N^2 ,\end{aligned}$$

using the binomial expansion theorem as in Claim 11.29. Plugging this in to Equation (11.8) we get the desired result:

$$\begin{aligned}f(a_1, b_1) \cdot f(a_2, b_2) &= (1+N)^{a_1 + a_2 \bmod N} \cdot (b_1 b_2 \bmod N)^N \bmod N^2 \\ &= f(a_1 + a_2, b_1 b_2),\end{aligned}$$

proving that f is an isomorphism from $\mathbb{Z}_N \times \mathbb{Z}_N^*$ to $\mathbb{Z}_{N^2}^*$. ■

11.3.2 The Paillier Encryption Scheme

Let $N = pq$ be a product of two distinct primes of equal length. Proposition 11.27 says that $\mathbb{Z}_N \times \mathbb{Z}_N^*$ is isomorphic to $\mathbb{Z}_{N^2}^*$, with isomorphism given by $f(a, b) = [(1+N)^a \cdot b^N \bmod N^2]$. A consequence is that a random element $y \in \mathbb{Z}_{N^2}^*$ corresponds to a random element $(a, b) \in \mathbb{Z}_N \times \mathbb{Z}_N^*$ or, in other words, an element (a, b) with random $a \in \mathbb{Z}_N$ and random $b \in \mathbb{Z}_N^*$.

Say $y \in \mathbb{Z}_{N^2}^*$ is an Nth *residue modulo* N^2 if y is an Nth power; that is, if there exists an $x \in \mathbb{Z}_{N^2}^*$ with $y = x^N \bmod N^2$. We denote the set of Nth residues modulo N^2 by $\mathsf{Res}(N^2)$. Let us characterize the Nth residues in $\mathbb{Z}_{N^2}^*$. Taking any $x \in \mathbb{Z}_{N^2}^*$ with $x \leftrightarrow (a, b)$ and raising it to the Nth power gives:

$$[x^N \bmod N^2] \leftrightarrow (a, b)^N = (N \cdot a \bmod N,\ b^N \bmod N) = (0, b^N \bmod N)$$

(recall that the group operation in $\mathbb{Z}_N \times \mathbb{Z}_N^*$ is addition modulo N in the first component and multiplication modulo N in the second component). Moreover, we claim that any element y with $y \leftrightarrow (0, b)$ is an Nth residue. To see this, recall that $\gcd(N, \phi(N)) = 1$ and so $d \stackrel{\text{def}}{=} [N^{-1} \bmod \phi(N)]$ exists. So

$$(a,\ [b^d \bmod N])^N = (Na \bmod N, [b^{dN} \bmod N]) = (0, b) \leftrightarrow y$$

for any $a \in \mathbb{Z}_N$. We have thus shown that $\mathsf{Res}(N^2)$ corresponds to the set

$$\{(0, b) \mid b \in \mathbb{Z}_N^*\} .$$

(Compare this to $\mathbb{Z}_{N^2}^*$, which corresponds to $\{(a, b) \mid a \in \mathbb{Z}_N,\ b \in \mathbb{Z}_N^*\}$.) The above also demonstrates that the number of Nth roots of any $y \in \mathsf{Res}(N^2)$

is exactly N, and so computing Nth powers is an N-to-1 function. As a consequence, if $r \leftarrow \mathbb{Z}^*_{N^2}$ is chosen uniformly at random then $[r^N \bmod N^2]$ is a uniformly distributed element of $\mathsf{Res}(N^2)$.

The *decisional composite residuosity problem*, roughly speaking, is to distinguish a random element of $\mathbb{Z}^*_{N^2}$ from a random element of $\mathsf{Res}(N^2)$. Note that unlike the quadratic residuosity assumption, where $\mathcal{QR}_N$ and $\mathcal{QNR}_N^{+1}$ are disjoint sets, here $\mathsf{Res}(N^2) \subseteq \mathbb{Z}^*_{N^2}$. Nevertheless, $\mathsf{Res}(N^2)$ forms only a negligible fraction of $\mathbb{Z}^*_{N^2}$. Formally, let GenModulus be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$, and p and q are n-bit primes (except with probability negligible in n). Then:

DEFINITION 11.31 *We say* the decisional composite residuosity problem is hard relative to GenModulus *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\left| \Pr[\mathcal{A}(N, [r^N \bmod N^2]) = 1] - \Pr[\mathcal{A}(N, r) = 1] \right| \leq \mathsf{negl}(n),$$

where in each case the probabilities are taken over the experiment in which $\mathsf{GenModulus}(1^n)$ *outputs (N, p, q), and then a random $r \leftarrow \mathbb{Z}^*_{N^2}$ is chosen. (Note that $[r^N \bmod N^2]$ is a random element of* $\mathsf{Res}(N^2)$.)

The *decisional composite residuosity* (*DCR*) *assumption* is the assumption that there exists a GenModulus relative to which the decisional composite residuosity problem is hard. This assumption can be viewed as a generalization, of sorts, of the quadratic residuosity assumption in $\mathbb{Z}^*_N$ that we have seen earlier.

As we have discussed, random elements of $\mathbb{Z}^*_{N^2}$ have the form (r', r) with r' and r random (in the appropriate groups), while random Nth residues have the form $(0, r)$ with r random. The DCR assumption is that it is hard to distinguish random elements of the first type from random elements of the second type. This suggests the following abstract way to encrypt a message $m \in \mathbb{Z}_N$ with respect to a public key N: choose a random Nth residue $(0, r)$ and set the ciphertext equal to

$$c \leftrightarrow (m, 1) \cdot (0, r) = (m, r).$$

(In the above, '$\cdot$' represents the group operation in $\mathbb{Z}_N \times \mathbb{Z}^*_N$.) Without worrying for now how this can be carried out efficiently by the sender, or how the receiver can decrypt, let us simply convince ourselves (on an intuitive level) that this is secure. Since a random Nth residue $(0, r)$ cannot be distinguished from a random element (r', r), the ciphertext as constructed above is indistinguishable (from the point of an eavesdropper who does not know the factorization of N and so supposedly cannot compute f^{-1}) from a ciphertext constructed as

$$c' \leftrightarrow (m, 1) \cdot (r', r) = ([m + r' \bmod N], r)$$

for a random $r' \in \mathbb{Z}_N$. Lemma 10.20 shows that $[m + r' \bmod N]$ is uniformly distributed in $\mathbb{Z}_N$ and so, in particular, this ciphertext c' is independent of the message m. Indistinguishability of encryptions in the presence of an eavesdropper follows.

A formal proof that proceeds exactly along these lines is given below. Let us first see how encryption and decryption can be performed efficiently, and then give a formal description of the encryption scheme.

Encryption. We have described encryption above as though it is taking place in $\mathbb{Z}_N \times \mathbb{Z}_N^*$. In fact it takes place in the isomorphic group $\mathbb{Z}_{N^2}^*$. That is, the sender generates a ciphertext $c \in \mathbb{Z}_{N^2}^*$ by choosing a random $r \in \mathbb{Z}_N^*$ and then computing

$$c := [(1+N)^m \cdot r^N \bmod N^2].$$

Observe that

$$c = \big((1+N)^m \cdot 1^N\big) \cdot \big((1+N)^0 \cdot r^N\big) \bmod N^2 \leftrightarrow (m,1) \cdot (0,r),$$

and so $c \leftrightarrow (m,r)$ as desired.

We remark that it does not make any difference whether the sender chooses random $r \leftarrow \mathbb{Z}_N^*$ or random $r \leftarrow \mathbb{Z}_{N^2}^*$, since in either case the distribution of $[r^N \bmod N^2]$ is the same (as can be verified by looking at what happens in the isomorphic group $\mathbb{Z}_N \times \mathbb{Z}_N^*$).

Decryption. We now describe how decryption can be performed efficiently given the factorization of N. For c constructed as above, we claim that m is recovered by the following steps:

- Set $\hat{c} := [c^{\phi(N)} \bmod N^2]$.
- Set $\hat{m} := (\hat{c} - 1)/N$. (Note that this is carried out over the integers.)
- Set $m := \big[\hat{m} \cdot \phi(N)^{-1} \bmod N\big]$.

To see why this works, let $c \leftrightarrow (m,r)$ for an arbitrary $r \in \mathbb{Z}_N^*$. Then

$$\begin{aligned}
\hat{c} &\stackrel{\text{def}}{=} [c^{\phi(N)} \bmod N^2] \\
&\leftrightarrow (m,\, r)^{\phi(N)} \\
&= \Big([m \cdot \phi(N) \bmod N],\ [r^{\phi(N)} \bmod N]\Big) \\
&= \big([m \cdot \phi(N) \bmod N],\ 1\big).
\end{aligned}$$

By Proposition 11.27(3), this means that $\hat{c} = (1+N)^{[m \cdot \phi(N) \bmod N]} \bmod N^2$. Using Proposition 11.27(2), we know that

$$\hat{c} = (1+N)^{[m \cdot \phi(N) \bmod N]} = (1 + [m \cdot \phi(N) \bmod N] \cdot N) \bmod N^2.$$

Since $1+[m\cdot\phi(N) \bmod N]\cdot N$ is always less that N^2 we can drop the $\bmod N^2$ at the end and view the above as an equality over the integers. Thus, $\hat{m} \stackrel{\text{def}}{=} (\hat{c}-1)/N = [m \cdot \phi(N) \bmod N]$ and, finally,

$$m = [\hat{m} \cdot \phi(N)^{-1} \bmod N],$$

as required. (Note that $\phi(N)$ is invertible modulo N since $\gcd(N, \phi(N)) = 1$.)

We give a complete description of the Paillier encryption scheme, followed by an example of the above calculations.

CONSTRUCTION 11.32

Let GenModulus be a polynomial-time algorithm that, on input 1^n, outputs (N, p, q) where $N = pq$ and p and q are n-bit primes (except with probability negligible in n). Define a public-key encryption scheme as follows:

- Gen: on input 1^n run GenModulus(1^n) to obtain (N, p, q). The public key is N, and the private key is $\langle N, \phi(N)\rangle$.
- Enc: on input a public key N and a message $m \in \mathbb{Z}_N$, choose a random $r \leftarrow \mathbb{Z}_N^*$ and output the ciphertext

$$c := [(1+N)^m \cdot r^N \bmod N^2].$$

- Dec: on input a private key $\langle N, \phi(N)\rangle$ and a ciphertext c, compute

$$m := \left[\frac{[c^{\phi(N)} \bmod N^2] - 1}{N} \cdot \phi(N)^{-1} \bmod N\right].$$

The Paillier encryption scheme.

Example 11.33

Let $N = 11 \cdot 17 = 187$ (and so $N^2 = 34969$), and consider encrypting the message $m = 175$ and then decrypting the corresponding ciphertext. Choosing $r = 83 \in \mathbb{Z}_{187}^*$, we compute the ciphertext

$$c := [(1+187)^{175} \cdot 83^{187} \bmod 34969] = 23911$$

corresponding to $(175, 83)$. To decrypt, note that $\phi(N) = 160$. So we first compute $\hat{c} := [23911^{160} \bmod 34969] = 25620$. Subtracting 1 and dividing by 187 gives

$$\hat{m} := (25620 - 1)/187 = 137;$$

since $90 = [160^{-1} \bmod 187]$, the message is recovered as

$$m := [137 \cdot 90 \bmod 187] = 175.$$

$\diamondsuit$

Correctness follows from the earlier discussion. We now prove security.

THEOREM 11.34 *If the decisional composite residuosity problem is hard relative to* GenModulus*, then the Paillier encryption scheme has indistinguishable encryptions under a chosen-plaintext attack.*

PROOF Let Π denote the Paillier encryption scheme. We prove that Π has indistinguishable encryptions in the presence of an eavesdropper; by Theorem 10.10 this implies that it is CPA-secure.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1].$$

Consider the following PPT adversary D that attempts to solve the decisional composite residuosity problem relative to GenModulus:

> **Algorithm** D:
> The algorithm is given N, y as input.
>
> - Set $pk = \langle N \rangle$ and run $\mathcal{A}(pk)$ to obtain two messages m_0, m_1.
> - Choose a random bit b and set $c := [(1+N)^{m_b} \cdot y \bmod N^2]$.
> - Give the ciphertext c to $\mathcal{A}$ and obtain an output bit b'. If $b' = b$, output 1; otherwise, output 0.

Let us analyze the behavior of D. There are two cases to consider:

Case 1: Say the input to D was generated by running $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q), choosing random $r \leftarrow \mathbb{Z}^*_{N^2}$, and setting $y := [r^N \bmod N^2]$. (That is, y is a random element of $\mathsf{Res}(N^2)$.) In this case, the ciphertext c is constructed as

$$c = [(1+N)^{m_b} \cdot r^N \bmod N^2]$$

for a random $r \in \mathbb{Z}^*_{N^2}$. Recalling that the distribution on $[r^N \bmod N^2]$ is the same whether r is chosen at random from $\mathbb{Z}^*_N$ or from $\mathbb{Z}^*_{N^2}$, we see that in this case the view of $\mathcal{A}$ when run as a subroutine by D is distributed exactly as $\mathcal{A}$'s view in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. Since D outputs 1 exactly when the output b' of $\mathcal{A}$ is equal to b, we have that

$$\Pr[D(N, [r^N \bmod N^2]) = 1] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)\,,$$

where the left-most probability is taken over the appropriate experiment in Definition 11.31.

Case 2: Say the input to D was generated by running $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q) and choosing random $y \leftarrow \mathbb{Z}^*_{N^2}$. We claim that the view of $\mathcal{A}$ in this case is *independent* of the bit b. To see this, note that since y is a

random element of the group $\mathbb{Z}^*_{N^2}$, the ciphertext c is randomly distributed in $\mathbb{Z}^*_{N^2}$ (see Lemma 10.20) and, in particular, is independent of m. This means the probability that $b' = b$ in this case is exactly $\frac{1}{2}$. That is,

$$\Pr[D(N, r) = 1] = \frac{1}{2},$$

where the probability is taken over the appropriate experiment in Definition 11.31.

Combining the above we have that

$$\left| \Pr[D(N, [r^N \bmod N^2]) = 1] - \Pr[D(N, r) = 1] \right| \leq \left| \varepsilon(n) - \tfrac{1}{2} \right|.$$

By the assumption that the decisional composite residuosity problem is hard relative to $\mathsf{GenModulus}$, there exists a negligible function negl such that

$$\left| \varepsilon(n) - \tfrac{1}{2} \right| \leq \mathsf{negl}(n)$$

and so $\varepsilon(n) \leq \frac{1}{2} + \mathsf{negl}(n)$, completing the proof. ■

11.3.3 Homomorphic Encryption

The Paillier encryption scheme turns out to be useful in a number of contexts since it is an example of a *homomorphic* encryption scheme over an *additive* group. That is, if we let $\mathsf{Enc}_N(m)$ denote the (randomized) Paillier encryption of a message $m \in \mathbb{Z}_N$ with respect to the public key N, we have

$$\mathsf{Enc}_N(m_1) \cdot \mathsf{Enc}_N(m_2) = \mathsf{Enc}_N([m_1 + m_2 \bmod N])$$

for all $m_1, m_2 \in \mathbb{Z}_N$. To see this, one can verify that

$$\begin{aligned} &\left((1+N)^{m_1} \cdot r_1^N\right) \cdot \left((1+N)^{m_2} \cdot r_2^N\right) \\ &= (1+N)^{[m_1+m_2 \bmod N]} \cdot (r_1 r_2)^N \bmod N^2, \end{aligned}$$

and the latter is a valid encryption of the message $[m_1 + m_2 \bmod N]$.

DEFINITION 11.35 *A public-key encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* $\mathsf{homomorphic}$ *if for all* n *and all* (pk, sk) *output by* $\mathsf{Gen}(1^n)$*, it is possible to define groups* $\mathbb{M}, \mathbb{C}$ *such that:*

- *The plaintext space is* $\mathbb{M}$*, and all ciphertexts output by* Enc_{pk} *are elements of* $\mathbb{C}$*.*
- *For any* $m_1, m_2 \in \mathbb{M}$ *and* $c_1, c_2 \in \mathbb{C}$ *with* $m_1 = \mathsf{Dec}_{sk}(c_1)$ *and* $m_2 = \mathsf{Dec}_{sk}(C_2)$*, it holds that*

  $$\mathsf{Dec}_{sk}(c_1 \cdot c_2) = m_1 \cdot m_2,$$

  where the group operations are carried out in $\mathbb{C}$ *and* $\mathbb{M}$*, respectively.*

Restating what we have said above, the Paillier encryption scheme is homomorphic taking $\mathbb{M} = \mathbb{Z}_N$ and $\mathbb{C} = \mathbb{Z}_{N^2}^*$ for the public key $pk = N$.

The Paillier encryption scheme is not the first homomorphic encryption scheme we have seen. El Gamal encryption is also homomorphic: if $\mathsf{Gen}(1^n)$ outputs $pk = (\mathbb{G}, q, g, h)$ then messages are elements of $\mathbb{G}$ and ciphertexts are elements of $\mathbb{G} \times \mathbb{G}$. Furthermore,

$$\langle g^{y_1},\ h^{y_1} \cdot m_1 \rangle \cdot \langle g^{y_2},\ h^{y_2} \cdot m_2 \rangle = \langle g^{y_1+y_2},\ h^{y_1+y_2} \cdot m_1 m_2 \rangle,$$

is a valid encryption of the message $m_1 m_2 \in \mathbb{G}$. The Goldwasser-Micali encryption is also homomorphic (see Exercise 11.17).

A nice feature of Paillier encryption is that it is homomorphic over a large *additive* group (namely, $\mathbb{Z}_N$). To see why this might be useful, imagine the following cryptographic voting scheme based on Paillier encryption:

1. An authority generates a public key N for the Paillier encryption scheme and publicizes N.

2. Let 0 stand for a "no", and let 1 stand for a "yes". Each of ℓ voters can cast their vote by encrypting it. That is, voter i casts her vote v_i by computing $c_i := [(1+N)^{v_i} \cdot r^N \bmod N^2]$ for a randomly-chosen $r \leftarrow \mathbb{Z}_N^*$.

3. Each voter broadcasts their vote c_i. These votes are then publicly *aggregated* by computing

$$c^* := \left[\prod_{i=1}^{\ell} c_i \bmod N^2 \right].$$

   Any external observer can verify that this step was done properly.

4. The authority is given the ciphertext c^*. (We assume the authority has not been able to observe what goes on until now.) By decrypting c^*, the authority obtains the vote total

$$v^* = \sum_{i=1}^{\ell} v_i \bmod N.$$

   If ℓ is small (so that $v^* \ll N$), then $v^* = \sum_{i=1}^{\ell} v_i$.

Key features of the above are that the authority obtains the correct vote total *without learning any individual votes*. Furthermore, *no voter learns anyone else's vote, either*, and calculation of the vote total is *publicly verifiable*. We assume here that all parties act honestly (and only try to learn others' votes based on the information they have observed). There are attacks on the above protocol when this assumption does not hold, and an entire research area of cryptography is dedicated to formalizing appropriate security notions in settings such as these, and designing secure protocols.

References and Additional Reading

Childs [32] and Shoup [131] provide further coverage of the (computational) number theory used in this chapter. A good description of the algorithm for computing the Jacobi symbol modulo a composite of unknown factorization, along with a proof of correctness, is given in [46]. The problem of deciding quadratic residuosity modulo a composite of unknown factorization goes back to Gauss [60] and is related to other (conjectured) hard number-theoretic problems. The Goldwasser-Micali encryption scheme is from [69], and was the first public-key encryption scheme with a rigorous proof of security.

Rabin [119] showed that computing square roots modulo a composite is equivalent to factoring. See [28] for indication that solving the RSA problem is not equivalent to factoring. The method shown in Section 11.2.2 for obtaining a family of permutations based on squaring modulo a composite is due to Blum [21]. Hard-core predicates for the Rabin trapdoor permutation are discussed in [5, 75, 4] and references therein.

Section 10.2 of Shoup's text [131] characterizes $\mathbb{Z}^*_{N^e}$ for arbitrary integers N, e (and not just $N = pq$, $e = 2$ as done here). The Paillier encryption scheme was introduced in [113].

Exercises

11.1 Let $\mathbb{G}$ be an abelian group. Show that the set of quadratic residues in $\mathbb{G}$ forms a subgroup.

11.2 This question concerns the quadratic residues in the additive group $\mathbb{Z}_N$. (An element $y \in \mathbb{Z}_N$ is a quadratic residue if and only if there exists an $x \in \mathbb{Z}_N$ with $2x = y \bmod N$.)

(a) What are the quadratic residues in $\mathbb{Z}_p$ for p an odd prime?

(b) Let $N = pq$ be a product of two odd primes p and q. What are the quadratic residues in $\mathbb{Z}_N$?

(c) Let N be an even integer. What are the quadratic residues in $\mathbb{Z}_N$?

11.3 Let $N = pq$ with p, q distinct, odd primes. Show a PPT algorithm for choosing a random element of $\mathcal{QNR}^{+1}_N$ *when the factorization of* N *is known.* (Your algorithm can have failure probability negligible in $\|N\|$.)

11.4 Let $N = pq$ with p, q distinct, odd primes. Prove that if $x \in \mathcal{QR}_N$ then $[x^{-1} \bmod N] \in \mathcal{QR}_N$, and if $x \in \mathcal{QNR}^{+1}_N$ then $[x^{-1} \bmod N] \in \mathcal{QNR}^{+1}_N$.

11.5 Let $N = pq$ with p, q distinct, odd primes, and fix $z \in \mathcal{QNR}_N^{+1}$. Show that choosing random $x \leftarrow \mathcal{QR}_N$ and setting $y := [z \cdot x \bmod N]$ gives a y that is uniformly distributed in $\mathcal{QNR}_N^{+1}$. I.e., for any $\hat{y} \in \mathcal{QNR}_N^{+1}$

$$\Pr[z \cdot x = \hat{y} \bmod N] = 1/|\mathcal{QNR}_N^{+1}|,$$

where the probability is taken over random choice of $x \leftarrow \mathcal{QR}_N$.

Hint: Use the previous exercise.

11.6 Let N be the product of 5 distinct odd primes. If $y \in \mathbb{Z}_N^*$ is a quadratic residue, how many solutions are there to the equation $x^2 = y \bmod N$?

11.7 Consider the following variation of the Goldwasser-Micali encryption scheme: $\mathsf{GenModulus}(1^n)$ is run to obtain (N, p, q) where $N = pq$ and $p = q = 3 \bmod 4$. (I.e., N is a Blum integer.) The public key is N and the secret key is $\langle p, q\rangle$. To encrypt $m \in \{0, 1\}$, the sender chooses random $x \in \mathbb{Z}_N$ and computes the ciphertext $c := [(-1)^m \cdot x^2 \bmod N]$.

(a) Prove that for N of the stated form, $[-1 \bmod N] \in \mathcal{QNR}_N^{+1}$.

(b) Prove that the scheme described has indistinguishable encryptions under a chosen-plaintext attack if deciding quadratic residuosity is hard relative to $\mathsf{GenModulus}$.

11.8 Assume deciding quadratic residuosity is hard for $\mathsf{GenModulus}$. Show that this implies the hardness of distinguishing a random element of $\mathcal{QR}_N$ from a random element of $\mathcal{J}_N^{+1}$.

11.9 Consider the following variation of the Goldwasser-Micali encryption scheme: $\mathsf{GenModulus}(1^n)$ is run to obtain (N, p, q). The public key is N and the secret key is $\langle p, q\rangle$. To encrypt a 0, the sender chooses n random elements $c_1, \ldots, c_n \leftarrow \mathcal{QR}_N$. To encrypt a 1, the sender chooses n random elements $c_1, \ldots, c_n \leftarrow \mathcal{J}_N^{+1}$. In each case, the resulting ciphertext is $c^* = \langle c_1, \ldots, c_n\rangle$.

(a) Show how the sender can generate a random element of $\mathcal{J}_N^{+1}$ in polynomial time.

(b) Suggest a way for the receiver to decrypt efficiently, though with error probability negligible in n.

(c) Prove that if deciding quadratic residuosity is hard relative to $\mathsf{GenModulus}$, then this scheme is CPA-secure.

Hint: Use the previous exercise.

11.10 Let $\mathcal{G}$ be a polynomial-time algorithm that, on input 1^n, outputs a prime p with $\|p\| = n$ and a generator g of $\mathbb{Z}_p^*$. Prove that the DDH problem is *not* hard relative to $\mathcal{G}$.

Hint: Use the fact that quadratic residuosity can be decided efficiently modulo a prime.

11.11 The discrete logarithm problem is believed to be hard for $\mathcal{G}$ as in the previous exercise. This means that the function (family) $f_{p,g}$ where $f_{p,g}(x) \stackrel{\text{def}}{=} [g^x \bmod p]$ is one-way. Let $\mathsf{lsb}(x)$ denote the least-significant bit of x. Show that lsb is *not* a hard-core predicate for $f_{p,g}$.

11.12 Consider a "textbook Rabin" encryption scheme in which a message $m \in \mathcal{QR}_N$ is encrypted relative to a public key N (where N is a Blum integer) by computing the ciphertext $c := [m^2 \bmod N]$. Show a chosen-ciphertext attack on this scheme that recovers the entire private key.

11.13 Let N be a Blum integer.

(a) Define the set $S \stackrel{\text{def}}{=} \{x \in \mathbb{Z}_N^* \mid x < N/2 \text{ and } \mathcal{J}_N(x) = 1\}$. Define the function $f_N : S \to \mathbb{Z}_N^*$ by:

$$f_N(x) = \begin{cases} [x^2 \bmod N] & \text{if } [x^2 \bmod N] < N/2 \\ [-x^2 \bmod N] & \text{if } [x^2 \bmod N] > N/2 \end{cases}$$

Show that f_N is a permutation over S.

(b) Define a family of trapdoor permutations based on factoring using f_N as defined above.

11.14 (a) Let N be a Blum integer. Define the function $\mathsf{half}_N : \mathbb{Z}_N^* \to \{0,1\}$ as

$$\mathsf{half}_N(x) = \begin{cases} 0 \text{ if } x < N/2 \\ 1 \text{ if } x > N/2 \end{cases}$$

Show that the function $f : \mathbb{Z}_N^* \to \mathcal{QR}_N \times \{0,1\}^2$ defined as

$$f(x) = [x^2 \bmod N], \mathcal{J}_N(x), \mathsf{half}_N(x)$$

is one-to-one.

(b) Suggest a "padded Rabin" encryption scheme that encrypts messages of length n. (All algorithms of your scheme should run in polynomial time, and the scheme should have correct decryption. Although a proof of security is unlikely, your scheme should not be susceptible to any obvious attacks.)

11.15 Show that the isomorphism of Lemma 11.27 can be efficiently inverted when the factorization of N is known.

11.16 Let $\Phi(N^2)$ denote the set $\{(a,1) \mid a \in \mathbb{Z}_N\} \subset \mathbb{Z}_{N^2}^*$. Show that it is *not* hard to decide whether a given element $y \in \mathbb{Z}_{N^2}^*$ is in $\Phi(N^2)$.

11.17 Show that the Goldwasser-Micali encryption scheme is homomorphic when the plaintext space $\{0,1\}$ is viewed as the group $\mathbb{Z}_2$.

Chapter 12

Digital Signature Schemes

12.1 Digital Signatures – An Overview

Thus far, we have dealt only with methods for achieving *private* communication in the public-key setting. We now turn to the question of preserving message *integrity*. The public-key counterpart of message authentication codes are *digital signatures*, though there are some important differences as we shall see below.

Digital signature schemes allow a *signer* S who has established a public key pk to "sign" a message in such a way that any other party who knows pk (and knows that this public key was established by S) can *verify* that the message originated from S and has not been modified in any way. As an example of typical usage of a digital signature scheme, consider a software company that wants to disseminate software patches in an authenticated manner: that is, when the company needs to release a software patch it should be possible for any of its clients to recognize that the patch is authentic, and a malicious third party should never be able to fool a client into accepting a patch that was not actually released by the company. To do this, the company can generate a public key pk along with a private key sk, and then distribute pk in some reliable manner to its clients while keeping sk secret. (As in the case of public-key encryption, we assume that this initial distribution of the public key is carried out correctly so that all clients have a correct copy of pk. In the current example, pk can be included with the original software purchased by a client.) When releasing a software patch m, the company can then compute a digital signature σ on m using its private key sk, and send the pair (m, σ) to every client. Each client can verify the authenticity of m by checking that σ is a legitimate signature on m with respect to the public key pk. The *same* public key pk is used by all clients, and so only a single signature needs to be computed by the company and sent to everyone.

A malicious party might try to issue a fake patch by sending (m', σ') to a client, where m' represents a patch that was never released by the company. This m' might be a modified version of some previous patch m, or it might be completely new and unrelated to previous patches. However, if the signature scheme is "secure" (in a sense we will define more carefully soon), then when the client attempts to verify σ' it will find that this is an *invalid* signature

on m' with respect to pk, and will therefore *reject* the signature. The client should reject even if m' is modified only slightly from a genuine patch m.

The above is not just a theoretical application of digital signatures, but one that is used extensively today.

Comparison to Message Authentication Codes

Both message authentication codes and digital signature schemes are used to ensure the integrity (or authenticity) of transmitted messages. Although the discussion in Chapter 9 comparing the public- and private-key settings focused on the case of encryption, it is easy to see that the discussion applies also to the case of message integrity. Use of digital signatures rather than message authentication codes simplifies key management, especially when a sender needs to communicate with multiple receivers. In particular, by using a digital signature scheme the sender can avoid having to establish a distinct secret key with each potential receiver, and avoid having to compute a separate MAC tag with respect to each such key. Instead, the sender need only compute a single signature that can be verified by all recipients.

A *qualitative* advantage that digital signatures have as compared to message authentication codes is that signatures are *publicly verifiable*. This means that if a receiver verifies the signature on a given message as being legitimate, then it is assured that all other parties who receive this signed message will also verify it as legitimate. This feature is not achieved by message authentication codes where a signer shares a separate key with each receiver: in such a setting a malicious sender might compute a correct MAC tag with respect to receiver A's shared key but an incorrect MAC tag with respect to a different user B's shared key. In this case, A knows that he received an authentic message from the sender but has no guarantee that other recipients will agree.

Public verifiability implies that signatures are *transferable*: a signature σ on a message m by a particular signer S can be shown to a third party, who can then verify herself that σ is a legitimate signature on m with respect to S's public key (here, we assume this third party also knows S's public key). By making a copy of the signature, this third party can then show the signature to another party and convince *them* that S authenticated m, and so on. Transferability and public verifiability are essential for the application of digital signatures to certificates and public-key infrastructures, as we will discuss in further detail in Section 12.8.

Digital signature schemes also provide the very important property of *non-repudiation*. That is — assuming a signer S widely publicizes his public key in the first place — once S signs a message he cannot later deny having done so. This aspect of digital signatures is crucial for situations where a recipient needs to prove to a third party (say, a judge) that a signer did indeed "certify" a particular message (e.g., a contract): assuming S's public key is known to the judge, or is otherwise publicly available, a valid signature on a message is enough to convince the judge that S indeed signed this message. Message

authentication codes simply cannot provide this functionality. To see this, say users S and R share a key k_{SR}, and S sends a message m to R along with a (valid) MAC tag tag computed using k_{SR}. Since the judge does *not* know k_{SR} (indeed, this key is kept secret by S and R), there is no way for the judge to determine whether tag is valid or not. If R were to reveal the key k_{SR} to the judge, there would still be no way for the judge to know whether this is the "actual" key that S and R shared, or whether it is some "fake" key manufactured by R. Even if we assume the judge is given the actual key k_{SR}, and can somehow be convinced of this fact, there is no way for R to prove that it was S who generated tag — the very fact that message authentication codes are *symmetric* (so that anything S can do, R can also do) implies that R could have generated tag on its own, and so there is no way for the judge to distinguish between the actions of the two parties.

As in the case of private- vs. public-key encryption, message authentication codes have the advantage of being roughly 2–3 orders of magnitude more efficient than digital signatures. Thus, in situations where public verifiability, transferability, and/or non-repudiation are not needed, and the sender will communicate primarily with a single recipient (with whom it is able to share a secret key), message authentication codes are preferred.

Relation to Public-Key Encryption

Digital signatures are often mistakenly viewed as the "inverse" of public-key encryption, with the roles of the sender and receiver interchanged. Historically, in fact, it has been suggested that digital signatures can be obtained by "reversing" public-key encryption, i.e., signing a message m by decrypting it (using the private key) to obtain σ, and verifying a signature σ by encrypting it (using the corresponding public key) and checking whether the result is m.[1] The suggestion to construct signature schemes in this way is *completely unfounded*: in most cases, it is simply inapplicable, and in cases when it is applicable it results in signature schemes that are completely insecure.

12.2 Definitions

As we have noted, digital signatures are the public-key counterpart of message authentication codes. The algorithm that the sender applies to a message is now denoted Sign (rather than Mac), and the output of this algorithm is

[1] This view no doubt arises in part because, as we will see in Section 12.3.1, "textbook RSA" signatures are indeed the reverse of textbook RSA encryption. However, neither textbook RSA signatures nor textbook RSA encryption meet even minimal notions of security.

now called a *signature* (rather than a tag). The algorithm that the receiver applies to a message and a signature in order to verify legitimacy is again denoted Vrfy. We now formally define the syntax of a digital signature scheme.

DEFINITION 12.1 *A* signature scheme *is a tuple of three probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *satisfying the following:*

1. *The* key-generation algorithm Gen *takes as input a security parameter* 1^n *and outputs a pair of keys* (pk, sk)*. These are called the* public key *and the* private key*, respectively. We assume for convenience that* pk *and* sk *each have length at least* n*, and that* n *can be determined from* pk, sk*.*

2. *The* signing algorithm Sign *takes as input a private key* sk *and a message*[2] $m \in \{0,1\}^*$*. It outputs a signature* σ*, denoted as* $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$*.*

3. *The deterministic* verification algorithm Vrfy *takes as input a public key* pk*, a message* m*, and a signature* σ*. It outputs a bit* b*, with* $b = 1$ *meaning* valid *and* $b = 0$ *meaning* invalid*. We write this as* $b := \mathsf{Vrfy}_{pk}(m, \sigma)$*.*

It is required that for every n*, every* (pk, sk) *output by* $\mathsf{Gen}(1^n)$*, and every* $m \in \{0,1\}^*$*, it holds that*

$$\mathsf{Vrfy}_{pk}(m, \mathsf{Sign}_{sk}(m)) = 1.$$

If $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *is such that for every* (pk, sk) *output by* $\mathsf{Gen}(1^n)$*, algorithm* Sign_{sk} *is only defined for messages* $m \in \{0,1\}^{\ell(n)}$ *(and* Vrfy_{pk} *outputs 0 for* $m \notin \{0,1\}^{\ell(n)}$*), then we say that* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *is a* signature scheme for messages of length $\ell(n)$*.*

We say that σ is a *valid signature* on a message m (with respect to some public key pk that is understood from the context) if $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$. We remark that the correctness requirement in the definition can be relaxed to allow for a negligible probability error (like in Definition 10.1). Although this is sometimes needed in concrete schemes, we ignore this issue in our presentation.

A signature scheme is used in the following way. One party S, who acts as the *sender*, runs $\mathsf{Gen}(1^n)$ to obtain keys (pk, sk). The public key pk is then publicized as belonging to S; e.g., S can put the public key on its webpage or place it in some public directory. As in the case of public-key encryption, we assume that any other party is able to obtain a legitimate copy of S's public key (see discussion below). When S wants to transmit a message m, it computes the signature $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and sends (m, σ). Upon receipt of

[2] As in the case of public-key encryption, we could also assume some underlying message space that may depend on pk. Except for the textbook RSA signature scheme (which is anyway insecure) all schemes in this book will sign bit strings.

(m, σ), a receiver who knows pk can verify the authenticity of m by checking whether $\mathsf{Vrfy}_{pk}(m, \sigma) \stackrel{?}{=} 1$. This establishes both that S sent m, and also that m was not modified in transit. As in the case of message authentication codes, however, it does not say anything about *when* m was sent, and replay attacks are still possible (see Section 4.3).

The assumption that parties are able to obtain a legitimate copy of S's public key implies that S is able to transmit at least one message (namely, pk itself) in a reliable and authenticated manner. Given this, one may wonder why signature schemes are needed at all! The point is that reliable distribution of pk *is* a difficult task, but using a signature scheme means that this need only be carried out *once*, after which an unlimited number of messages can subsequently be sent reliably. Furthermore, as we will discuss in Section 12.8, signature schemes themselves are used to ensure the reliable distribution of *other* public keys. As we will see, this is a central tool in the "public-key infrastructure" solution to the key distribution problem.

Security of signature schemes. Given a public key pk generated by a signer S, we say that an adversary outputs a *forgery* if it outputs a message m along with a valid signature σ on m, and furthermore m was not previously signed by S. As in the case of message authentication, security of a digital signature scheme means that an adversary cannot output a forgery even if it is allowed to obtain signatures on many other messages of its choice. This is the direct analogue of the definition of security for message authentication codes, and we refer the reader to Section 4.3 for motivation and further discussion.

Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme, and consider the following experiment for an adversary $\mathcal{A}$ and parameter n:

The signature experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n)$**:**

1. $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
2. *Adversary* $\mathcal{A}$ *is given* pk *and oracle access to* $\mathsf{Sign}_{sk}(\cdot)$. *(This oracle returns a signature* $\mathsf{Sign}_{sk}(m)$ *for any message* m *of the adversary's choice.) The adversary then outputs* (m, σ). *Let* $\mathcal{Q}$ *denote the set of messages whose signatures were requested by* $\mathcal{A}$ *during its execution.*
3. *The output of the experiment is defined to be 1 if and only if* **(1)** $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$, *and* **(2)** $m \notin \mathcal{Q}$.

We now present the definition of security, which is essentially the same as Definition 4.2 for message authentication codes.

DEFINITION 12.2 *A signature scheme* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *is* existentially unforgeable under an adaptive chosen-message attack *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

12.3 RSA Signatures

We begin our consideration of concrete signature schemes with a discussion of various schemes based on the RSA assumption. We warn the reader that *none of the schemes in this section are known to be secure* (though a variant of one of the schemes we show here will later be proven secure in a model that is discussed in detail in Chapter 13). We introduce these schemes mainly to provide examples of attacks on digital signature schemes, as well as to provide some intuition as to why constructing secure signature schemes is highly non-trivial.

12.3.1 "Textbook RSA" and its Insecurity

We begin by introducing the "textbook RSA" signature scheme. We refer to the scheme as such since many textbooks describe RSA signatures in exactly this way with no further warning. Unfortunately, "textbook RSA" signatures are *insecure* as we will demonstrate below.

Let GenRSA be a PPT algorithm that, on input 1^n, outputs a modulus N that is the product of two n-bit primes (except with negligible probability), along with integers e, d satisfying $ed = 1 \bmod \phi(N)$. Key generation in the textbook RSA scheme is performed by simply running GenRSA, and outputting $\langle N, e\rangle$ as the public key and $\langle N, d\rangle$ as the private key. To sign a message $m \in \mathbb{Z}_N^*$, the signer computes $\sigma := [m^d \bmod N]$. Verification of a signature σ on a message m with respect to the public key $\langle N, e\rangle$ is carried out by checking whether $m \stackrel{?}{=} \sigma^e \bmod N$. See Construction 12.3.

CONSTRUCTION 12.3

Let GenRSA be as in the text. Define a signature scheme as follows:

- Gen: on input 1^n run $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d). The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.
- Sign: on input a private key $sk = \langle N, d\rangle$ and a message $m \in \mathbb{Z}_N^*$, compute the signature
$$\sigma := [m^d \bmod N].$$
- Vrfy: on input a public key $pk = \langle N, e\rangle$, a message $m \in \mathbb{Z}_N^*$, and a signature $\sigma \in \mathbb{Z}_N^*$, output 1 if and only if
$$m \stackrel{?}{=} [\sigma^e \bmod N].$$

The "textbook RSA" signature scheme.

It is easy to see that verification of a legitimately-generated signature is always successful since

$$\sigma^e = (m^d)^e = m^{[ed \bmod \phi(N)]} = m^1 = m \bmod N.$$

The textbook RSA signature scheme is insecure, however, as the following examples demonstrate.

A no-message attack. It is trivial to output a forgery for the textbook RSA signature scheme *based on the public key alone*, without even obtaining any signatures from the legitimate signer. The attack works as follows: given a public key $pk = \langle N, e \rangle$, choose an arbitrary $\sigma \in \mathbb{Z}_N^*$ and compute $m := [\sigma^e \bmod N]$. Then output the forgery (m, σ). It is immediate that σ is a valid signature on m, and this is obviously a forgery since no signature on m (in fact, no signatures at all!) was generated by the owner of the public key. We conclude that the textbook RSA signature scheme does not satisfy Definition 12.2.

One may argue that this does not constitute a "realistic" attack since the adversary has "no control" over the message m for which it forges a valid signature. Of course, this is irrelevant as far as Definition 12.2 is concerned, and we have already discussed (in Chapter 4) why it is dangerous to assume any semantics for messages that are going to be authenticated using any cryptographic scheme. Moreover, the adversary does have *some* control over m: for example, by choosing multiple random values of σ it can (with high probability) obtain an m with certain bits set in any desired way. Or, by choosing σ in some structured manner, rather than at random, it may also be possible to influence the message for which a forgery can be output.

Forging a signature on an arbitrary message. A more damaging attack on the textbook RSA signature scheme requires the adversary to obtain *two* signatures from the signer, but allows the adversary to output a forgery on any message of the adversary's choice. Say the adversary wants to forge a signature on the message $m \in \mathbb{Z}_N^*$ with respect to the public key $pk = \langle N, e \rangle$. The adversary chooses a random $m_1 \in \mathbb{Z}_N^*$, sets $m_2 := [m/m_1 \bmod N]$, and then obtains signatures σ_1 and σ_2 on m_1 and m_2, respectively. We claim that $\sigma := [\sigma_1 \cdot \sigma_2 \bmod N]$ is a valid signature on m. This is because

$$\sigma^e = (\sigma_1 \cdot \sigma_2)^e = (m_1^d \cdot m_2^d)^e = m_1^{ed} \cdot m_2^{ed} = m_1 m_2 = m \bmod N,$$

using the fact that σ_1, σ_2 are valid signatures on m_1, m_2. This constitutes a forgery since m is not equal to m_1 or m_2 (except with negligible probability).

Being able to forge a signature on an arbitrary message is clearly devastating. Nevertheless, one might argue that this attack is unrealistic since an adversary will never be able to convince a signer to sign the exact messages m_1 and m_2 as needed for the above attack. Once again, this is irrelevant as far as Definition 12.2 is concerned. Furthermore, it is dangerous to make assumptions about what messages the signer will or will not be willing to sign. For

example, signing may be used as a method of authentication whereby a client proves its identity by signing a random value sent by a server. A malicious server could then obtain a signature on any message(s) of its choice. It may also be possible for the adversary to choose m_1 in a particular way (rather than at random) in order to make m_2 a "legitimate" message that the signer will sign. Note also that the attack can be generalized: if an adversary obtains valid signatures on some q arbitrary messages $M = \{m_1, \ldots, m_q\}$, then the adversary can output a forgery for any of $2^q - q - 1$ messages obtained by taking products of subsets of M (of size greater than 1).

12.3.2 Hashed RSA

Numerous modifications of the textbook RSA signature scheme have been proposed in an effort to prevent the attacks described in the previous section. Most of these proposals have not been proven secure, and should not be used. Nevertheless, we show one such example here (another example is given in Exercise 12.4). This example serves as an illustration of a more general paradigm that will be explored in the following section, and can be proven secure in an "idealized" model that will be described in detail in Chapter 13.

The basic idea is to modify the textbook RSA signature scheme by applying some function H to the message before signing it. That is, the public and private keys are the same as before except that a description of some function $H : \{0,1\}^* \to \mathbb{Z}_N^*$ is now included as part of the public key. A message $m \in \{0,1\}^*$ is signed by computing

$$\sigma := [H(m)^d \bmod N].$$

(That is, $\hat{m} := H(m)$ is first computed, followed by $\sigma := [\hat{m}^d \bmod N]$.) Verification of the pair (m, σ) is carried out by checking whether

$$\sigma^e \stackrel{?}{=} H(m) \bmod N.$$

Clearly, verification of a legitimately-generated signature will always succeed.

An immediate observation is that a minimal requirement for the above scheme to be secure is that H must be collision-resistant (see Section 4.6): if it is not, and an adversary can find two different messages m_1, m_2 with $H(m_1) = H(m_2)$, then forgery is trivial. (Note, however, that H need not be compressing.) Since H must be a collision-resistant hash function, this modified scheme described is sometimes called the *hashed RSA signature scheme*.

There is no known function H for which hashed RSA signatures are known to be secure in the sense of Definition 12.2. Nevertheless, we can at least describe the intuition as to why the attacks shown on the textbook RSA signature scheme in the previous section are likely to be more difficult to carry out against the hashed RSA signature scheme.

The no-message attack. The natural way to attempt the no-message attack shown previously is to choose an arbitrary $\sigma \in \mathbb{Z}_N^*$, compute $\hat{m} := [\sigma^e \bmod N]$, and then try to find some $m \in \{0,1\}^*$ such that $H(m) = \hat{m}$. If the function H is not efficiently invertible this appears difficult to do.

Forging a signature on an arbitrary message. The natural way to attempt the chosen-message attack shown previously requires the adversary to find three messages m, m_1, m_2 for which $H(m) = [H(m_1) \cdot H(m_2) \bmod N]$. Once again, if H is not efficiently invertible this seems difficult to do.

Security of hashed RSA. Variants of hashed RSA are used in practice. It is possible to prove the security of hashed RSA in an idealized model where H is a *truly random* function. Although H is clearly *not* random (because H is a concrete, deterministic function), this provides a heuristic justification of the scheme when H is a "random-looking" cryptographic hash function such as SHA-1. See Chapter 13 for further discussion regarding the idealized model just mentioned, and the meaning of security proofs in this model.

12.4 The "Hash-and-Sign" Paradigm

The hashed RSA signature scheme can be viewed as an attempt to prevent certain attacks on the textbook RSA signature scheme. The success of this approach is mixed, as discussed above. We may note, however, that hashed RSA offers another advantage relative to textbook RSA: it can be used to sign *arbitrary-length bit-strings*, rather than just elements of $\mathbb{Z}_N^*$. This feature is useful in general, and the approach of hashing a message and then signing the result (using an underlying signature scheme that is secure) is a standard way to achieve it. We study the security of this approach now.

Let $\Pi = (\mathsf{Gen}_S, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme for messages of length $\ell(n) = n$. (Everything that follows can be modified appropriately for an arbitrary ℓ, as long as it is super-logarithmic in n.) Let $\Pi_H = (\mathsf{Gen}_H, H)$ be a hash function as per Definition 4.11, where the output of H, on security parameter 1^n, has length n. We construct a signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy}')$ for arbitrary-length messages as follows: $\mathsf{Gen}'(1^n)$ computes $(pk, sk) \leftarrow \mathsf{Gen}_S(1^n)$ and $s \leftarrow \mathsf{Gen}_H(1^n)$, and sets the public key equal to $\langle pk, s\rangle$ and the private key equal to $\langle sk, s\rangle$. To sign the message $m \in \{0,1\}^*$, the signer simply computes $\sigma \leftarrow \mathsf{Sign}_{sk}(H^s(m))$. Verification is performed by checking that $\mathsf{Vrfy}_{pk}(H^s(m), \sigma) \stackrel{?}{=} 1$. See Construction 12.4. Note that the hashed RSA signature scheme is indeed constructed from the textbook RSA signature scheme using exactly this approach (of course, the theorem below does not apply in that case because textbook RSA signatures are insecure).

CONSTRUCTION 12.4

Let $\Pi = (\mathsf{Gen}_S, \mathsf{Sign}, \mathsf{Vrfy})$ and $\Pi_H = (\mathsf{Gen}_H, H)$ be as in the text. Construct a signature scheme Π' for arbitrary-length messages as follows:

- Gen': on input 1^n, run $\mathsf{Gen}_S(1^n)$ to obtain (pk, sk), and run $\mathsf{Gen}_H(1^n)$ to obtain s. The public key is $pk' = \langle pk, s\rangle$ and the private key is $sk' = \langle sk, s\rangle$.
- Sign': on input a private key $\langle sk, s\rangle$ and a message $m \in \{0,1\}^*$, compute $\sigma' \leftarrow \mathsf{Sign}_{sk}(H^s(m))$.
- Vrfy': on input a public key $\langle pk, s\rangle$, a message $m \in \{0,1\}^*$, and a signature σ, output 1 if and only if $\mathsf{Vrfy}_{pk}(H^s(m), \sigma) \stackrel{?}{=} 1$.

The hash-and-sign paradigm.

THEOREM 12.5 *If Π is existentially unforgeable under an adaptive chosen-message attack and Π_H is collision resistant, then Construction 12.4 is existentially unforgeable under an adaptive chosen-message attack.*

PROOF The idea behind the proof is that a forgery must involve either finding a collision in H or forging a signature with respect to the fixed-length signature scheme Π. This intuition was described immediately following Theorem 4.16 in Section 4.7, so we proceed directly to the formal proof here.

Let $\mathcal{A}'$ be a probabilistic polynomial-time adversary attacking Π'. In an execution of experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n)$, let $pk' = \langle pk, s\rangle$ denote the public key used, let $\mathcal{Q}$ denote the set of messages whose signatures were requested by $\mathcal{A}'$, and let (m, σ) be the final output of $\mathcal{A}'$. We assume without loss of generality that $m \notin \mathcal{Q}$. Define $\mathsf{coll}_{\mathcal{A}',\Pi'}(n)$ to be the event that, in experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n)$, there exists an $m' \in \mathcal{Q}$ for which $H^s(m') = H^s(m)$.

We have

$$\begin{aligned}
&\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1] \\
&= \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \mathsf{coll}_{\mathcal{A}',\Pi'}(n)] \\
&\quad + \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\mathsf{coll}}_{\mathcal{A},\Pi'}(n)] \\
&\leq \Pr[\mathsf{coll}_{\mathcal{A}',\Pi'}(n)] + \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\mathsf{coll}}_{\mathcal{A},\Pi'}(n)]. \qquad (12.1)
\end{aligned}$$

We show that both terms in Equation (12.1) are negligible, which completes the proof. Intuitively, the first term is negligible by the collision resistance of Π_H, and the second term is negligible by the security of Π. (In order to see why the security of Π implies that the second term is negligible, notice that when $\overline{\mathsf{coll}}_{\mathcal{A},\Pi'}(n)$ occurs, $H^s(m) \neq H^s(m')$ for every $m' \in \mathcal{Q}$. This implies that a successful forgery (m, σ) by $\mathcal{A}'$ with respect to Π' yields a successful forgery $(H^s(m), \sigma)$ with respect to Π. By the security of Π, this must occur with negligible probability.) We begin by proving the first term.

Consider the following PPT algorithm $\mathcal{C}$ for finding a collision in Π_H:

Algorithm $\mathcal{C}$:
The algorithm is given s as input (with n implicit).

- Compute $\mathsf{Gen}_S(1^n)$ to obtain (pk, sk). Set $pk' = \langle pk, s\rangle$.
- Run $\mathcal{A}'$ on input pk'. When $\mathcal{A}'$ requests the ith signature on some message $m_i \in \{0,1\}^*$, compute $\sigma_i \leftarrow \mathsf{Sign}_{sk}(H^s(m_i))$ and give σ_i to $\mathcal{A}'$.
- Eventually, $\mathcal{A}'$ outputs (m, σ). If there exists an i for which $H^s(m) = H^s(m_i)$, output (m, m_i).

Let us analyze the behavior of $\mathcal{C}$. When the input to $\mathcal{C}$ is generated by running $\mathsf{Gen}_H(1^n)$ to obtain s, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{C}$ is distributed identically to the view of $\mathcal{A}'$ in experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. Since $\mathcal{C}$ outputs a collision exactly when $\mathsf{coll}_{\mathcal{A}',\Pi'}(n)$ occurs, we have

$$\Pr[\mathsf{Hash\text{-}coll}_{\mathcal{C},\Pi_H}(n) = 1] = \Pr[\mathsf{coll}_{\mathcal{A}',\Pi'}(n)].$$

Since Π_H is collision resistant, we conclude that $\Pr[\mathsf{coll}_{\mathcal{A}',\Pi'}(n)]$ is negligible.

We now proceed to prove that the second term in Equation (12.1) is negligible. Consider the following PPT adversary $\mathcal{A}$ attacking signature scheme Π:

Adversary $\mathcal{A}$:
The adversary is given as input a public key pk (with n implicit), and has access to a signing oracle $\mathsf{Sign}_{sk}(\cdot)$.

- Compute $\mathsf{Gen}_H(1^n)$ to obtain s, and set $pk' = \langle pk, s\rangle$.
- Run $\mathcal{A}'$ on input pk'. When $\mathcal{A}'$ requests the ith signature on a message $m_i \in \{0,1\}^*$, this is answered as follows: **(1)** compute $\hat{m}_i := H^s(m_i)$; then **(2)** obtain a signature σ_i on $\hat{m}_i$ from the signing oracle, and give σ_i to $\mathcal{A}'$.
- Eventually, $\mathcal{A}'$ outputs (m, σ). Output $(H^s(m), \sigma)$.

Consider experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n)$. In this experiment, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{A}$ is distributed exactly as its view in experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. Furthermore, it can be easily verified that whenever both $\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1$ and $\overline{\mathsf{coll}}_{\mathcal{A},\Pi'}(n)$ occur, $\mathcal{A}$ outputs a forgery. (It is clear that σ is a valid signature on $H^s(m)$ with respect to pk. The fact that $\overline{\mathsf{coll}}_{\mathcal{A},\Pi'}(n)$ occurs means that $\hat{m} = H^s(m)$ was never asked by $\mathcal{A}$ to its own signing oracle and so $\hat{m} \notin \mathcal{Q}$.) Therefore,

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A}',\Pi'}(n) \wedge \overline{\mathsf{coll}}_{\mathcal{A},\Pi'}(n)],$$

and security of Π implies that the latter probability is negligible. This concludes the proof of the theorem. ∎

An analogue of Theorem 12.5 holds for the case of message authentication codes, and gives an alternative to Theorem 4.6 for constructing variable-length MACs from fixed-length MACs (albeit under the additional assumption that collision-resistant hash functions exist, which is not needed for Theorem 4.6). As noted in Section 4.7, this "hash-and-MAC" approach is the basis for the security of NMAC and HMAC.

12.5 Lamport's One-Time Signature Scheme

Although Definition 12.2 is the standard definition of security for digital signature schemes, weaker definitions have also been considered. Signature schemes satisfying these weaker definitions may be appropriate for certain restricted applications, and may also serve as useful "building blocks" for signature schemes satisfying stronger notions of security, as we will see in the following section.

In this section, we define *one-time signature schemes* which, informally, are "secure" as long as they are used to sign only a *single* message. We then show a construction, due to Lamport, of a one-time signature scheme based on any one-way function. We begin with the definition. Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme, and consider the following experiment for an adversary $\mathcal{A}$ and parameter n:

The one-time signature experiment $\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n)$**:**

1. $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
2. *Adversary* $\mathcal{A}$ *is given* pk *and asks a* single *query* m' *to oracle* $\mathsf{Sign}_{sk}(\cdot)$. $\mathcal{A}$ *then outputs* (m, σ) *where* $m \neq m'$.
3. *The output of the experiment is defined to be 1 if and only if* $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$.

DEFINITION 12.6 *A signature scheme* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *is* existentially unforgeable under a single-message attack*, or is a* one-time signature scheme*, if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

The basic idea of Lamport's signature scheme is simple, and we illustrate it for the case of signing a 3-bit message. Let f be a one-way function. Recall this means f is easy to compute but hard to invert; see Definition 7.66. The public key consists of 6 elements $y_{1,0}$, $y_{1,1}$, $y_{2,0}$, $y_{2,1}$, $y_{3,0}$, $y_{3,1}$ in the range

Signing $m = 011$:

$$sk = \begin{pmatrix} \boxed{x_{1,0}} & x_{2,0} & x_{3,0} \\ x_{1,1} & \boxed{x_{2,1}} & \boxed{x_{3,1}} \end{pmatrix} \Rightarrow \sigma = (x_{1,0},\, x_{2,1},\, x_{3,1})$$

Verifying for $m = 011$ and $\sigma = (x_1, x_2, x_3)$:

$$pk = \left.\begin{pmatrix} \boxed{y_{1,0}} & y_{2,0} & y_{3,0} \\ y_{1,1} & \boxed{y_{2,1}} & \boxed{y_{3,1}} \end{pmatrix}\right\} \Rightarrow \begin{array}{l} f(x_1) \stackrel{?}{=} y_{1,0} \\ f(x_2) \stackrel{?}{=} y_{2,1} \\ f(x_3) \stackrel{?}{=} y_{3,1} \end{array}$$

FIGURE 12.1: The Lamport scheme used to sign the message $m = 011$.

of f; the private key contains the corresponding pre-images $x_{1,0}$, $x_{1,1}$, $x_{2,0}$, $x_{2,1}$, $x_{3,0}$, $x_{3,1}$. These keys can be visualized as two-dimensional arrays:

$$pk = \begin{pmatrix} y_{1,0} & y_{2,0} & y_{3,0} \\ y_{1,1} & y_{2,1} & y_{3,1} \end{pmatrix} \quad sk = \begin{pmatrix} x_{1,0} & x_{2,0} & x_{3,0} \\ x_{1,1} & x_{2,1} & x_{3,1} \end{pmatrix}.$$

To sign a message $m = m_1 \cdot m_2 \cdot m_3$, where each m_i is a single bit, the signer releases the appropriate pre-image x_{i,m_i} for $1 \le i \le 3$; the signature σ simply consists of the three values $(x_{1,m_1}, x_{2,m_2}, x_{3,m_3})$. Verification is carried out in the natural way: presented with the candidate signature (x_1, x_2, x_3) on the message $m = m_1 \cdot m_2 \cdot m_3$, accept if and only if $f(x_i) \stackrel{?}{=} y_{i,m_i}$ for $1 \le i \le 3$. This is shown graphically in Figure 12.1. The general case for an arbitrary length ℓ is described formally in Construction 12.7.

CONSTRUCTION 12.7

Let f be a one-way function. Construct a signature scheme for messages of length $\ell = \ell(n)$ as follows:

- Gen: on input 1^n, proceed as follows for $i \in \{1, \ldots, \ell\}$:
  1. Choose random $x_{i,0}, x_{i,1} \leftarrow \{0,1\}^n$.
  2. Compute $y_{i,0} := f(x_{i,0})$ and $y_{i,1} := f(x_{i,1})$.

  The public key pk and the private key sk are

$$pk := \begin{pmatrix} y_{1,0} & y_{2,0} & \cdots & y_{\ell,0} \\ y_{1,1} & y_{2,1} & \cdots & y_{\ell,1} \end{pmatrix} \quad sk := \begin{pmatrix} x_{1,0} & x_{2,0} & \cdots & x_{\ell,0} \\ x_{1,1} & x_{2,1} & \cdots & x_{\ell,1} \end{pmatrix}.$$

- Sign: on input a private key sk as above and a message $m \in \{0,1\}^\ell$ with $m = m_1 \cdots m_\ell$, output the signature $(x_{1,m_1}, \ldots, x_{\ell,m_\ell})$.
- Vrfy: on input a public key pk as above, a message $m \in \{0,1\}^\ell$ with $m = m_1 \cdots m_\ell$, and a signature $\sigma = (x_1, \ldots, x_\ell)$, output 1 if and only if $f(x_i) = y_{i,m_i}$ for all $1 \le i \le \ell$.

Lamport's signature scheme.

THEOREM 12.8 *Let ℓ be any polynomial. If f is a one-way function, then Construction 12.7 is a one-time signature scheme for messages of length ℓ.*

PROOF We let $\ell = \ell(n)$ for the rest of the proof. As intuition for the security of the scheme, note that for an adversary given public key $pk = \begin{pmatrix} y_{1,0} & y_{2,0} & \cdots & y_{\ell,0} \\ y_{1,1} & y_{2,1} & \cdots & y_{\ell,1} \end{pmatrix}$, finding an x such that $f(x) = y_{i^*,b^*}$ for *any* (i^*, b^*) amounts to inverting f. So it will certainly be hard to compute a signature on any message m given only the public key. What about computing a signature on some message m after being given a signature on a different message m'? If $m' \neq m$ then there must be at least one position i^* on which these messages differ. Say $m_{i^*} = b^* \neq m'_{i^*}$. Then forging a signature on m requires, in particular, finding an x such that $f(x) = y_{i^*,b^*}$. But finding such an x does not become any easier even when given $\{x_{i,b}\}$ for *all* $(i,b) \neq (i^*, b^*)$ (since the values $\{x_{i,b}\}_{(i,b)\neq(i^*,b^*)}$ are all chosen independently of x_{i^*,b^*}), and a signature on m' reveals even fewer "x-values" than these.

We now turn this intuition into a formal proof. Let Π denote the Lamport scheme. Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n) = 1].$$

In a particular execution of $\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n)$, let m' denote the message whose signature is requested by $\mathcal{A}$ (we assume without loss of generality that $\mathcal{A}$ always requests a signature on a message), and let (m, σ) denote the final output of $\mathcal{A}$. We say that $\mathcal{A}$ *output a forgery at* (i, b) if $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$ and furthermore $m_i \neq m'_i$ (i.e., messages m and m' differ on their ith position) and $m_i = b \neq m'_i$. Note that whenever $\mathcal{A}$ outputs a forgery, it outputs a forgery at *some* (i, b).

Consider the following PPT algorithm $\mathcal{I}$ attempting to invert the one-way function f:

Algorithm $\mathcal{I}$:
The algorithm is given y and 1^n as input.

1. Choose random $i^* \leftarrow \{1, \ldots, \ell\}$ and $b^* \leftarrow \{0,1\}$. Set $y_{i^*,b^*} := y$.
2. For all $i \in \{1, \ldots, \ell\}$ and $b \in \{0,1\}$ with $(i,b) \neq (i^*, b^*)$:
   - Choose $x_{i,b} \leftarrow \{0,1\}^n$ and set $y_{i,b} := f(x_{i,b})$.
3. Run $\mathcal{A}$ on input $pk := \begin{pmatrix} y_{1,0} & y_{2,0} & \cdots & y_{\ell,0} \\ y_{1,1} & y_{2,1} & \cdots & y_{\ell,1} \end{pmatrix}$.
4. When $\mathcal{A}$ requests a signature on the message m':
   - If $m'_{i^*} = b^*$, stop.
   - Otherwise, return the correct signature $\sigma = (x_{1,m'_1}, \ldots, x_{\ell,m'_\ell})$.
5. When $\mathcal{A}$ outputs (m, σ) with $\sigma = (x_1, \ldots, x_p)$:
   - If $\mathcal{A}$ output a forgery at (i^*, b^*), then output x_{i^*}.

Whenever $\mathcal{A}$ outputs a forgery at (i^*, b^*), algorithm $\mathcal{I}$ succeeds in inverting its given input y. We are interested in the probability that this occurs when the input to $\mathcal{I}$ is generated by choosing a random $x \leftarrow \{0,1\}^n$ and setting $y := f(x)$ (cf. Definition 7.66). Imagine a "mental experiment" in which $\mathcal{I}$ is given x at the outset, sets $x_{i^*,b^*} := x$, and then always returns a signature to $\mathcal{A}$ in step 4 (i.e., even if $m'_{i^*} = b^*$). It is not hard to see that the view of $\mathcal{A}$ being run as a subroutine by $\mathcal{I}$ in this mental experiment is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n)$. Therefore, the probability that $\mathcal{A}$ outputs a forgery in step 5 is exactly $\varepsilon(n)$. Because (i^*, b^*) was chosen at random at the beginning of the experiment, and the view of $\mathcal{A}$ is independent of this choice, the probability that $\mathcal{A}$ outputs a forgery at (i^*, b^*), conditioned on the fact that $\mathcal{A}$ outputs a forgery at all, is at least $1/2\ell(n)$. (This is because a signature forgery implies a forgery for at least one point (i, b). Since there are $2\ell(n)$ points, the probability of the forgery being at (i^*, b^*) is at least $1/2\ell(n)$.) We conclude that, in this mental experiment, the probability that $\mathcal{A}$ outputs a forgery at (i^*, b^*) is at least $\varepsilon(n)/2\ell(n)$.

Returning to the real experiment involving $\mathcal{I}$ as initially described, the key observation is that *the probability that $\mathcal{A}$ outputs a forgery at (i^*, b^*) is unchanged*. This is because the mental experiment and the real experiment coincide if $\mathcal{A}$ outputs a forgery at (i^*, b^*). That is, the experiments only differ if $\mathcal{A}$ requests a signature on a message m' with $m'_{i^*} = b^*$; but if this happens then it is impossible (by definition) for $\mathcal{A}$ to subsequently output a forgery at (i^*, b^*). So, in the real experiment, the probability that $\mathcal{A}$ outputs a forgery at (i^*, b^*) is still at least $\varepsilon(n)/2\ell(n)$. We thus have:

$$\Pr[\mathsf{Invert}_{\mathcal{I},f}(n) = 1] \geq \varepsilon(n)/2\ell(n).$$

Because f is a one-way function, there is a negligible function negl such that

$$\Pr[\mathsf{Invert}_{\mathcal{I},f}(n) = 1] \leq \mathsf{negl}(n).$$

This implies that $\varepsilon(n)/2\ell(n) \leq \mathsf{negl}(n)$, and since ℓ is polynomial, that $\varepsilon(n)$ itself is negligible. This completes the proof. ■

COROLLARY 12.9 *If one-way functions exist, then for any polynomial ℓ there exists a one-time signature scheme for messages of length ℓ.*

12.6 * Signatures from Collision-Resistant Hashing

We have not yet seen any signature scheme that is existentially unforgeable under an adaptive chosen-message attack (cf. Definition 12.2). Here we show a relatively inefficient construction that is essentially the simplest one known based on any of the cryptographic assumptions we have introduced thus far. The construction relies on the existence of collision-resistant hash functions, and serves mainly as a proof of feasibility for realizing Definition 12.2.

We remark that signature schemes satisfying Definition 12.2 are, in general, quite difficult to construct, and even today only a few *efficient* schemes that can be proven to satisfy this definition are known. In Chapter 13, we will discuss a very efficient signature scheme that can be proven secure in a certain "idealized" model that is introduced and discussed extensively there. The development of other efficient signature schemes that can be proven secure in the "standard" model we have been using until now is an area of active research today.

We build up to our final construction in stages. In Section 12.6.1 we define the notion of a *stateful* signature scheme, where the signer updates its private key after each signature, and show how to construct a stateful signature scheme that satisfies Definition 12.2. In Section 12.6.2 we discuss a more efficient variant of this scheme (that is still stateful) and show that this, too, is existentially unforgeable under an adaptive chosen-message attack. We then describe how this construction can be made stateless, so as to recover a signature scheme as originally defined.

12.6.1 "Chain-Based" Signatures

We first define signature schemes that allow the signer to maintain some *state* that is updated after every signature is produced.

DEFINITION 12.10 *A* stateful signature scheme *is a tuple of probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *satisfying the following:*

1. *The* key-generation algorithm Gen *takes as input a security parameter* 1^n *and outputs* (pk, sk, s_0)*. These are called the* public key*,* private key*, and* initial state*, respectively. We assume* pk *and* sk *each have length at least* n*, and that* n *can be determined from* pk, sk*.*

2. *The* signing algorithm Sign *takes as input a private key* sk*, a value* s_{i-1}*, and a message* $m \in \{0,1\}^*$*. It outputs a signature* σ *and a value* s_i*.*

3. *The deterministic* verification algorithm Vrfy *takes as input a public key* pk*, a message* m*, and a signature* σ*. It outputs a bit* b*.*

We require that for every n*, every* (pk, sk, s_0) *output by* $\mathsf{Gen}(1^n)$*, and any messages* $m_1, \ldots, m_\ell \in \{0,1\}^*$*, if we compute* $(\sigma_i, s_i) \leftarrow \mathsf{Sign}_{sk, s_{i-1}}(m_i)$ *recursively for* $i \in \{1, \ldots, \ell\}$*, then for every* $i \in \{1, \ldots, \ell\}$*,*

$$\mathsf{Vrfy}_{pk}(m_i, \sigma_i) = 1.$$

If $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *is such that for every* (pk, sk) *output by* $\mathsf{Gen}(1^n)$*, algorithm* Sign_{sk} *is only defined for messages* $m \in \{0,1\}^{\ell(n)}$ *(and* Vrfy_{pk} *outputs 0 for* $m \notin \{0,1\}^{\ell(n)}$*), then we say* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *is a* stateful signature scheme for messages of length $\ell(n)$*.*

We emphasize that the state is not needed to verify a signature (in fact, depending on the scheme, the state may need to be kept secret in order for security to hold). Signature schemes which do not maintain state are called *stateless* to distinguish them from stateful schemes. Clearly, stateless schemes are preferable (although stateful schemes can still potentially be useful). Nevertheless, as mentioned, our aim in introducing stateful signatures is as a stepping stone to a full stateless construction.

Existential unforgeability under an adaptive chosen-message attack for the case of stateful signatures schemes is defined in a manner exactly analogous to Definition 12.2, with the only subtleties being that the signing oracle only returns the signature (and *not* the state), and the signing oracle updates the state appropriately each time it is invoked.

We can easily construct a stateful "ℓ-time" signature scheme that can sign $\ell = \ell(n)$ messages for any polynomial ℓ. (The notion of security here would be analogous to the definition of one-time signatures given earlier; we do not give a formal definition since our discussion here is only informal.) Such a construction works by simply letting the public key consist of ℓ independently-generated public keys for any one-time signature scheme, with the private key similarly constructed; i.e., set $pk := (pk_1, \ldots, pk_\ell)$ and $sk := (sk_1, \ldots, sk_\ell)$ where each (pk_i, sk_i) is an independently-generated key-pair for some one-time signature scheme. The state is a counter initially set to 1. To sign a message m using the private key sk and current state $s \leq \ell$, simply output $\sigma \leftarrow \mathsf{Sign}_{sk_s}(m)$ (that is, generate a one-time signature on m using the private key sk_s) and update the state to $s+1$. Since the initial state starts at 1, this means the ith message is signed using sk_i. Verification of a signature σ on a message m can be done by checking whether σ is a valid signature on m with respect to any of the $\{pk_i\}$.

Intuitively, this scheme is secure if used to sign ℓ messages since each private key is used to sign only a *single* message. Since ℓ may be an arbitrary polynomial, why doesn't this give us the solution we are looking for? The main drawback is that the scheme requires the upper bound ℓ on the number of messages that can be signed *to be fixed in advance*, at the time of key generation. (In particular, the scheme does not satisfy Definition 12.2.) This is a potentially severe limitation since once the upper bound is reached a new public key would have to be generated and distributed. We would like instead to have a single, fixed scheme that can support signing an *unbounded* number of messages. Another drawback of the scheme is the fact that it is not very efficient, since the public and private keys have length that is linear in the total number of messages that can be signed.

Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a one-time signature scheme. In the scheme we have just described, the signer runs ℓ invocations of Gen to obtain public keys $pk_1, \ldots, pk_\ell$, and includes each of these in its actual public key pk. The signer is then restricted to signing at most ℓ messages. We can do better by using a "chain-based scheme" in which the signer generates and certifies additional public keys on-the-fly as needed.

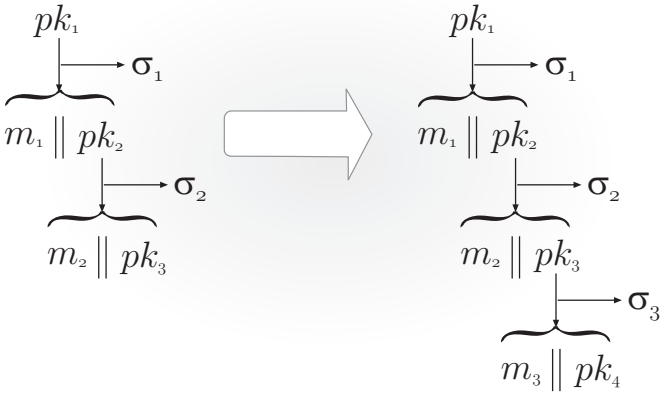

FIGURE 12.2: Chain-based signatures: the situation before and after signing the third message m_3.

In the chain-based scheme, the public key consists of just a single public key pk_1 generated using Gen, and the private key contains the associated private key sk_1. To sign the first message m_1, the signer first generates a new key-pair (pk_2, sk_2) using Gen, and then signs both m_1 and pk_2 using sk_1 to obtain $\sigma_1 \leftarrow \mathsf{Sign}_{sk_1}(m_1 \| pk_2)$. The signature that is output includes both pk_2 and σ_1, and the signer adds $(m_1, pk_2, sk_2, \sigma_1)$ to its current state. In general, when it comes time to sign the ith message the signer will have stored $\{(m_j, pk_{j+1}, sk_{j+1}, \sigma_j)\}_{j=1}^{i-1}$ as part of its state. To sign the ith message m_i, the signer first generates a new key-pair (pk_{i+1}, sk_{i+1}) using Gen, and then signs m_i and pk_{i+1} using sk_i to obtain a signature $\sigma_i \leftarrow \mathsf{Sign}_{sk_i}(m_i \| pk_{i+1})$. The actual signature that is output includes pk_{i+1}, σ_i, and also the values $\{m_j, pk_{j+1}, \sigma_j\}_{j=1}^{i-1}$. The signer then adds $(m_i, pk_{i+1}, sk_{i+1}, \sigma_i)$ to its state. See Figure 12.2 for a graphical depiction of this process.

To verify a signature $(pk_{i+1}, \sigma_i, \{m_j, pk_{j+1}, \sigma_j\}_{j=1}^{i-1})$ on a message $m = m_i$ with respect to public key pk_1, the receiver verifies each link between the public key pk_j and the next public key pk_{j+1} in the chain, as well as the link between the last public key pk_{i+1} and m. That is, the verification procedure outputs 1 if and only if $\mathsf{Vrfy}_{pk_j}(m_j \| pk_{j+1}, \sigma_j) \stackrel{?}{=} 1$ for all $j \in \{1, \ldots, i\}$. (Refer to Figure 12.2.) Observe that this verification begins from the public key pk_1 that was initially distributed.

It is not hard to be convinced — at least on an intuitive level — that a signature scheme thus constructed is existentially unforgeable under an adaptive chosen-message attack (regardless of how many messages are signed). Informally, this is once again due to the fact that each key-pair (pk_i, sk_i) is used to sign only a single "message" (where in this case the "message" is actually a message/public-key pair $m_i \| pk_{i+1}$). Since we are going to prove the security

of a more efficient scheme in the next section, we do not give a formal proof of security for the chain-based scheme here.

In the chain-based scheme, each public key pk_i is used to sign both a message and another public key. Thus, it is essential for the underlying one-time signature scheme Π to be capable of *signing messages longer than the public key*. The Lamport scheme presented in Section 12.5 does *not* have this property. However, if we apply the "hash-and-sign" paradigm from Section 12.4 to the Lamport scheme, we *do* obtain a one-time signature scheme that can sign messages of arbitrary length. (Although Theorem 12.5 was stated only with regard to signature schemes satisfying Definition 12.2, it is not hard to see that an identical proof works for one-time signature schemes.) Because this result is crucial for the next section, we state it formally.

LEMMA 12.11 *If collision-resistant hash functions exist, then there exists a one-time signature scheme (for messages of arbitrary length).*

PROOF As mentioned, we simply use the hash-and-sign paradigm of Theorem 12.5 in conjunction with the Lamport signature scheme. Note that the existence of collision-resistant hash functions implies the existence of one-way functions (see Exercise 12.10), and so this assumption suffices. ■

The chain-based signature scheme is a stateful signature scheme which is existentially unforgeable under an adaptive chosen-message attack. It has a number of disadvantages, though. For one, there is no immediate way to eliminate the state (recall that our ultimate goal is a stateless scheme satisfying Definition 12.2). It is also not very efficient, in that the signature length, size of the state, and verification time are all linear in the number of messages that have been signed. Finally, each signature reveals all previous messages that have been signed. While this does not technically violate any security requirement for signatures, this may be undesirable in some contexts.

12.6.2 "Tree-Based" Signatures

The signer in the chain-based scheme of the previous section can be viewed as maintaining a *tree*, rooted at the public key pk_1, whose degree is 1 and whose depth is equal to the number of messages signed thus far (cf. Figure 12.2). A natural way to improve the efficiency of this approach is to use a *binary* tree in which each node has degree 2. As before, a signature will correspond to a "certified" path in the tree from a leaf to the root; notice that as long as the tree has polynomial depth (even if it has exponential size!), verification can still be done in polynomial time.

Concretely, to sign messages of length n we will work with a binary tree of depth n having 2^n leaves. As before, the signer will add nodes to the tree

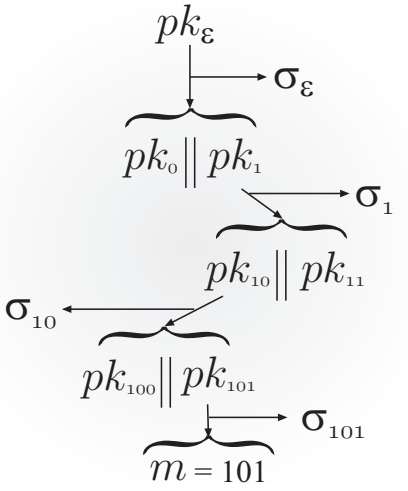

FIGURE 12.3: Tree-based signatures: signing the message $m = 101$.

"on-the-fly," as needed. In contrast to the chain-based scheme, though, only leaves (and not internal nodes) will be used to certify messages. Each leaf of the tree will correspond to one of the possible messages of length n.

In more detail, we imagine a binary tree of depth n where the root is labeled by ε (i.e., the empty string), and a node that is labeled with the binary string w of length less than n has left-child labeled $w0$ and right-child labeled $w1$. This tree is never constructed in its entirety (note that it is exponentially-large), but is instead built up by the signer as needed.

For every node w, we associate a pair of keys pk_w, sk_w for some one-time signature scheme Π. The public key of the root, pk_ε, is the actual public key of the signer. To sign a message $m \in \{0,1\}^n$, the signer carries out the following steps:

1. It first generates keys (as needed) for all nodes on the path from the root to the leaf labeled m. (Some of these public keys may have been generated in the process of signing previous messages, and in this case are not generated again.)

2. Next, it "certifies" the path from the root to the leaf labeled m by computing a signature on $pk_{w0} \| pk_{w1}$, using private key sk_w, for each string w that is a proper prefix of m.

3. Finally, it "certifies" m itself by computing a signature on m with the private key sk_m.

The final signature on m consists of the signature on m with respect to pk_m, as well as all the information needed to verify the path from the leaf labeled m to the root; see Figure 12.3. Additionally, the signer updates its state by storing all the key pairs generated as part of the above signing process. A formal description of this scheme is given as Construction 12.12.

CONSTRUCTION 12.12

Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme. For a binary string m, let $m|_i \stackrel{\text{def}}{=} m_1 \cdots m_i$ denote the i-bit prefix of m (with $m|_0 \stackrel{\text{def}}{=} \varepsilon$, the empty string). Construct the scheme $\Pi^* = (\mathsf{Gen}^*, \mathsf{Sign}^*, \mathsf{Vrfy}^*)$ as follows:

- Gen^*: on input 1^n, compute $(pk_\varepsilon, sk_\varepsilon) \leftarrow \mathsf{Gen}(1^n)$ and output the public key pk_ε. The private key and initial state are sk_ε.
- Sign^*: on input a message $m \in \{0,1\}^n$, carry out the following.
  1. For $i = 0$ to $n-1$:
     - If $pk_{m|_i 0}, pk_{m|_i 1}$, and $\sigma_{m|_i}$ are not in the state, compute $(pk_{m|_i 0}, sk_{m|_i 0}) \leftarrow \mathsf{Gen}(1^n)$, $(pk_{m|_i 1}, sk_{m|_i 1}) \leftarrow \mathsf{Gen}(1^n)$, and $\sigma_{m|_i} \leftarrow \mathsf{Sign}_{sk_{m|_i}}(pk_{m|_i 0} \,\|\, pk_{m|_i 1})$. In addition, add all of these values to the state.
  2. If σ_m is not yet included in the state, compute $\sigma_m \leftarrow \mathsf{Sign}_{sk_m}(m)$ and store it as part of the state.
  3. Output the signature $\left(\left\{\sigma_{m|_i}, pk_{m|_i 0}, pk_{m|_i 1}\right\}_{i=0}^{n-1}, \sigma_m\right)$.
- Vrfy^*: on input a public key pk_ε, message m, and signature $\left(\left\{\sigma_{m|_i}, pk_{m|_i 0}, pk_{m|_i 1}\right\}_{i=0}^{n-1}, \sigma_m\right)$, output 1 if and only if:
  1. $\mathsf{Vrfy}_{pk_{m|_i}}(pk_{m|_i 0} \,\|\, pk_{m|_i 1}, \sigma_{m|_i}) \stackrel{?}{=} 1$ for all $i \in \{0, \ldots, n-1\}$.
  2. $\mathsf{Vrfy}_{pk_m}(m, \sigma_m) \stackrel{?}{=} 1$.

A "tree-based" signature scheme.

Notice that each of the underlying keys in this scheme is used to sign only a *single* "message". Each key associated with an internal node signs a pair of public keys, and keys at leaves are used to sign a single message (once). Note that the keys are used to sign a pair of other keys, and thus we need the one-time signature scheme Π to be capable of signing messages longer than the public key. Lemma 12.11 shows that such schemes can be constructed based on collision-resistant hash functions.

Before proving security of this tree-based approach, note that it improves on the chain-based scheme in a number of respects. It still allows for signing an unbounded number of messages. (Although there are only 2^n leaves, the message space contains only 2^n messages. In any case, 2^n is eventually larger

than any polynomial function of n.) In terms of efficiency, the signature length and verification time are now proportional to the message length n but are *independent* of the number of messages signed. The scheme is still stateful, but we will see how this can be avoided after we prove the following result.

THEOREM 12.13 *Let Π be a one-time signature scheme. Then Construction 12.12 is existentially unforgeable under an adaptive chosen-message attack.*

PROOF Let Π^* denote Construction 12.12. Let $\mathcal{A}^*$ be a probabilistic polynomial time adversary, let $\ell^* = \ell^*(n)$ be a (polynomial) upper bound on the number of signing queries made by $\mathcal{A}^*$, and set $\ell(n) \stackrel{\text{def}}{=} 2n\ell^*(n) + 1$; for shorthand we will write ℓ instead of $\ell(n)$. Note that $\ell(n)$ upper bounds the number of public keys from Π that are needed to generate $\ell^*(n)$ signatures using Π^*. This is because each signature in π^* requires at most $2n$ new keys from Π (in the worst case), and one additional key from Π is used as the actual public key pk_ε. Define

$$\delta(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A}^*,\Pi^*}(n) = 1].$$

Consider the following PPT adversary $\mathcal{A}$ attacking the one-time signature scheme Π:

Adversary $\mathcal{A}$:
$\mathcal{A}$ is given as input a public key pk (the security parameter n is implicit).

- Choose a random index $i^* \leftarrow \{1, \ldots, \ell\}$. Construct a list $pk^1, \ldots, pk^\ell$ of keys as follows:
  - Set $pk^{i^*} := pk$.
  - For $i \neq i^*$, compute $(pk^i, sk^i) \leftarrow \mathsf{Gen}(1^n)$.
- Run $\mathcal{A}^*$ on input public key $pk_\varepsilon = pk^1$. When $\mathcal{A}^*$ requests a signature on a message m do:
  1. For $i = 0$ to $n - 1$:
     - If the values $pk_{m|_i 0}, pk_{m|_i 1}$, and $\sigma_{m|_i}$ have not yet been defined, then set $pk_{m|_i 0}$ and $pk_{m|_i 1}$ equal to the next two unused public keys pk^j and pk^{j+1}, and compute a signature $\sigma_{m|_i}$ on $pk_{m|_i 0} \,\|\, pk_{m|_i 1}$ with respect to $pk_{m|_i}$.[3]

[3] If $i \neq i^*$ then $\mathcal{A}$ can compute a signature with respect to pk^i by itself. $\mathcal{A}$ can also obtain a (single) signature with respect to pk^{i^*} by making the appropriate query to its signing oracle. This is what is meant here.

2. If σ_m is not yet defined, compute a signature σ_m on m with respect to pk_m (see footnote 3).
3. Give $\left(\left\{\sigma_{m|_i}, pk_{m|_i 0}, pk_{m|_i 1}\right\}_{i=0}^{n-1},\ \sigma_m\right)$ to $\mathcal{A}^*$.

- Say $\mathcal{A}^*$ outputs a message m (for which it had not previously requested a signature) and a signature $\left(\left\{\ \sigma'_{m|_i}, pk'_{m|_i 0}, pk'_{m|_i 1}\ \right\}_{i=0}^{n-1},\ \sigma'_m\right)$. If this is a valid signature on m, then:

  Case 1: Say there exists a $j \in \{0, \ldots, n-1\}$ for which $pk'_{m|_j 0} \neq pk_{m|_j 0}$ or $pk'_{m|_j 1} \neq pk_{m|_j 1}$; this includes the case when $pk_{m|_j 0}$ or $pk_{m|_j 1}$ were never defined by $\mathcal{A}$. Take the minimal such j, and let i be such that $pk^i = pk_{m|_j} = pk'_{m|_j}$ (such an i exists by the minimality of j). If $i = i^*$, output $(pk'_{m|_j 0} \| pk'_{m|_j 1},\ \sigma'_{m|_j})$.

  Case 2: If case 1 does not hold, then $pk'_m = pk_m$. Let i be such that $pk^i = pk_m$. If $i = i^*$, output (m, σ'_m).

This completes the description of $\mathcal{A}$.

In experiment $\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n)$, the view of $\mathcal{A}^*$ being run as a subroutine by $\mathcal{A}$ is distributed identically to the view of $\mathcal{A}^*$ in experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A}^*,\Pi^*}(n)$.[4] Thus, the probability that $\mathcal{A}^*$ outputs a forgery is exactly $\delta(n)$ when it is run as a subroutine by $\mathcal{A}$ in this experiment. Given that $\mathcal{A}^*$ outputs a forgery, consider each of the two possible cases described above:

Case 1: Since i^* was chosen uniformly at random and is independent of the view of $\mathcal{A}^*$, the probability that $i = i^*$ is exactly $1/\ell$. If $i = i^*$, then $\mathcal{A}$ requested a signature on the message $pk_{m|_j 0} \| pk_{m|_j 1}$ with respect to the public key $pk = pk^{i^*} = pk_{m|_j}$ that it was given (and requested no other signatures). Moreover,

$$pk'_{m|_j 0} \| pk'_{m|_j 1} \neq pk_{m|_j 0} \| pk_{m|_j 1}$$

and yet $\sigma'_{m|_j}$ is a valid signature on $pk'_{m|_j 0} \| pk'_{m|_j 1}$ with respect to pk. Thus, $\mathcal{A}$ outputs a forgery in this case.

Case 2: Again, since i^* was chosen uniformly at random and is independent of the view of $\mathcal{A}^*$, the probability that $i = i^*$ is exactly $1/\ell$. If $i = i^*$, then $\mathcal{A}$ did not request any signatures with respect to the public key $pk = pk^i = pk_m$ and yet σ'_m is a valid signature on m with respect to pk.

[4] As we have mentioned, $\mathcal{A}$ never "runs out" of public keys. A signing query of $\mathcal{A}^*$ uses $2n$ public keys; thus, even if new public keys were required to answer *every* signing query of $\mathcal{A}^*$ (which will in general not be the case), only $2n\ell^*(n)$ public keys would be needed by $\mathcal{A}$ in addition to the "root" public key pk_ε.

We see that, conditioned on $\mathcal{A}^*$ outputting a forgery (and regardless of which of the above cases occurs), $\mathcal{A}$ outputs a forgery with probability exactly $1/\ell$. This means that

$$\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n) = 1] = \delta(n)/\ell(n).$$

Because Π is a one-time signature scheme, we know that there exists a negligible function negl for which

$$\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

Since ℓ is polynomial, we conclude that $\delta(n)$ must be negligible. ■

A Stateless Solution

The signer's state in Π^* depends on the messages signed. However, it is possible to imagine having the signer generate all necessary information for all the nodes in the entire tree *in advance*, at the time of key generation. (That is, at the time of key generation the signer could generate the keys $\{(pk_w, sk_w)\}$ and the signatures $\{\sigma_w\}$ for all binary strings w of length at most n.) If key generation were done in this way, then the signer would not have to update its state at all; these values could all be stored as part of a (large) private key, and we would obtain a stateless scheme. The problem with this approach, of course, is that generating all these values would require *exponential* time.

An alternative is to store some *randomness* that can be used to generate the values $\{(pk_w, sk_w)\}$ and $\{\sigma_w\}$, as needed, rather than storing the values themselves. That is, the signer could store a random string r_w for each w, and whenever the values pk_w, sk_w are needed the signer can compute $(pk_w, sk_w) := \mathsf{Gen}(1^n; r_w)$, where this denotes the generation of a length-n key using random coins r_w. Similarly, if the signing procedure is probabilistic, the signer can store r'_w and then set $\sigma_w := \mathsf{Sign}_{sk_w}(pk_{w0} \| pk_{w1}; r'_w)$ (assuming here that $|w| < n$). Generating and storing sufficiently-many random strings, however, still requires exponential time and space.

A simple modification of this alternative gives a polynomial-time solution. Instead of storing random r_w and r'_w as suggested above, the signer can store two keys k, k' for a pseudorandom function F. When needed, the values pk_w, sk_w can now be generated by the following two-step process:

1. Compute $r_w := F_k(w)$.[5]

2. Compute $(pk_w, sk_w) := \mathsf{Gen}(1^n; r_w)$ (as before).

[5] We assume that the output length of F is sufficiently long, and that w is padded to some fixed-length string in a one-to-one fashion. We ignore these technicalities here.

In addition, the key k' is used to generate the value r'_w that is used to compute the signature σ_w. This gives a *stateless* signature scheme in which key generation (as well as signing and verifying) can be carried out in polynomial time. Intuitively, this works because storing a random function is equivalent to storing all the r_w and r'_w values that are needed, and storing a pseudorandom function is "just as good". We leave it as an exercise to prove that this modified scheme remains existentially unforgeable under an adaptive chosen-message attack.

Since the existence of collision-resistant hash functions implies the existence of one-way functions (cf. Exercise 12.10), and the latter implies the existence of pseudorandom functions (see Chapter 6), we have:

THEOREM 12.14 *If collision-resistant hash functions exist, then there exists a (stateless) signature scheme that is existentially unforgeable under an adaptive chosen-message attack.*

We remark that it is possible to construct signature schemes satisfying Definition 12.2 from the (minimal) assumption that one-way functions exist; a proof of this result is beyond the scope of this book.

12.7 The Digital Signature Standard (DSS)

Hashed RSA signatures as described in Section 12.3.2, and variants thereof, give one example of signature schemes that are widely used in practice. Another important example is the *Digital Signature Standard (DSS)*, sometimes also known as the Digital Signature Algorithm (DSA). This scheme was proposed by the US National Institute of Standards and Technology (NIST) in 1991, and has since become a US government standard. The security of DSS relies on the hardness of the discrete logarithm problem, and has been used for many years without any serious attacks being found. As in the case of hashed RSA signatures, however, there is no known proof of security for DSS based on the discrete logarithm (or any other) assumption. We noted previously that hashed RSA signatures *can* be proven secure in a certain idealized model to be described in the following chapter. Unfortunately, DSS has no proof of security even in this idealized model. For these reasons, we must content ourselves with only giving a description of the scheme.

Let $\mathcal{G}$ be a probabilistic polynomial-time algorithm that, on input 1^n, outputs (p, q, g) where, except with negligible probability: **(1)** p and q are primes with $\|q\| = n$; **(2)** $q \mid (p-1)$ but $q^2 \nmid (p-1)$; and **(3)** g is a generator of the subgroup of $\mathbb{Z}_p^*$ having order q. (E.g., the algorithm in Section 7.3.3 could be used.) DSS is given as Construction 12.15.

CONSTRUCTION 12.15

Let $\mathcal{G}$ be as in the text. Define a signature scheme as follows:

- Gen: on input 1^n, run the algorithm $\mathcal{G}(1^n)$ to obtain (p, q, g). Let $H : \{0,1\}^* \to \mathbb{Z}_q$ be function. Choose $x \leftarrow \mathbb{Z}_q$ uniformly at random and set $y := [g^x \bmod p]$. The public key is $\langle H, p, q, g, y\rangle$ and the private key is $\langle H, p, q, g, x\rangle$.
- Sign: on input a private key $\langle H, p, q, g, x\rangle$ and a message $m \in \{0,1\}^*$, choose $k \leftarrow \mathbb{Z}_q^*$ uniformly at random and set $r := [\,[g^k \bmod p\,] \bmod q]$. Compute $s := [(H(m) + xr) \cdot k^{-1} \bmod q]$, and output the signature (r, s).
- Vrfy: on input a public key $\langle H, p, q, g, y\rangle$, a message $m \in \{0,1\}^*$, and a signature (r, s) with $r \in \mathbb{Z}_q$ and $s \in \mathbb{Z}_q^*$, compute the values $u_1 := [H(m) \cdot s^{-1} \bmod q]$ and $u_2 := [r \cdot s^{-1} \bmod q]$. Output 1 if and only if
$$r \stackrel{?}{=} [[g^{u_1} y^{u_2} \bmod p] \bmod q].$$

The Digital Signature Standard (DSS).

Let us see that the scheme is correct. Letting $\hat{m} = H(m)$, the signature (r, s) output by the signer satisfies

$$r = [\,[g^k \bmod p\,] \bmod q] \quad \text{and} \quad s = [(\hat{m} + xr) \cdot k^{-1} \bmod q].$$

Assume $s \neq 0$ (this occurs with only negligible probability). Using the fact that $y = g^x$ and recalling that we can work "in the exponent" modulo q, we have

$$\begin{aligned} g^{\hat{m}s^{-1}} y^{rs^{-1}} &= g^{\hat{m}\cdot(\hat{m}+xr)^{-1}k} g^{xr\cdot(\hat{m}+xr)^{-1}k} \bmod p \\ &= g^{(\hat{m}+xr)\cdot(\hat{m}+xr)^{-1}k} \bmod p \\ &= g^k \bmod p\,. \end{aligned}$$

Thus, $[[g^{\hat{m}s^{-1}} y^{rs^{-1}} \bmod p\,] \bmod q] = [[g^k \bmod p\,] \bmod q] = r$, and verification succeeds.

12.8 Certificates and Public-Key Infrastructures

We conclude this chapter with a brief discussion of one of the primary applications of digital signatures: the secure distribution of public keys. This brings us full-circle in our discussion of public-key cryptography. In this and the previous three chapters we have seen how to *use* public-key cryptography once public keys are securely distributed. Now we show how public-key cryptography itself can be used to securely distribute public keys. This may sound

circular, but is not. Essentially what we will show is that once a *single* public key (belonging to some trusted party) is distributed in a secure fashion, this key can be used to "bootstrap" the secure distribution of arbitrarily-many other public keys. Thus, the problem of secure key distribution need only be solved *once* (theoretically speaking, at least).

The key idea is the notion of a *digital certificate*, which is simply a signature binding some entity to some public key. To be concrete, say a party Charlie has generated a key-pair (pk_C, sk_C) for a secure digital signature scheme (in this section, we will only be concerned with signature schemes satisfying Definition 12.2). Assume further that another party Bob has also generated a key-pair (pk_B, sk_B) (in the present discussion, these may be keys for either a signature scheme or a public-key encryption scheme), and that Charlie *knows* that pk_B is Bob's public key. Then Charlie can compute the signature

$$\mathsf{cert}_{C\to B} \stackrel{\text{def}}{=} \mathsf{Sign}_{sk_C}(\texttt{`Bob's key is } pk_B\texttt{'})$$

and give this signature to Bob. This signature $\mathsf{cert}_{C\to B}$ is called a *certificate* for Bob's key issued by Charlie. In practice a certificate should unambiguously identify the party holding a particular public key and so a more uniquely descriptive term than "Bob" would be used, for example, Bob's full name and email address.

Now say Bob wants to communicate with some other party Alice who already knows pk_C. What Bob can do is to send $(pk_B, \mathsf{cert}_{C\to B})$ to Alice, who can then verify that $\mathsf{cert}_{C\to B}$ is indeed a valid signature on the message `'Bob's key is` pk_B`'` with respect to pk_C. Assuming verification succeeds, Alice now knows that Charlie has signed the indicated message. If Alice trusts Charlie, then she might now accept pk_B as Bob's legitimate public key.

Note that all communication between Bob and Alice can occur over an *insecure* and *unauthenticated* channel. If an active adversary interferes with the communication of $(pk_B, \mathsf{cert}_{C\to B})$ from Bob to Alice, that adversary will be unable to generate a valid certificate linking Bob to any *other* public key pk'_B unless Charlie had previously signed some other certificate linking Bob with pk'_B (in which case this is anyway not much of an attack). This all assumes that Charlie is not dishonest and that his private signing key has not been compromised.

We have omitted many details in the above description. Most prominently, we have not discussed how Alice learns pk_C in the first place; how Charlie can be sure that pk_B is Bob's public key; and how Alice decides whether to trust Charlie in the first place. Fully specifying such details (and others) gives a *public-key infrastructure* (PKI) that enables the widespread distribution of public keys. A variety of different PKI models have been suggested, and we mention a few of the more popular ones now. Our treatment here will be kept at a relatively high level, and the reader interested in further details is advised to consult the references at the end of this chapter.

A single certificate authority. The simplest PKI assumes a single *certificate authority* (CA) who is completely trusted by everybody and who issues certificates for everyone's public key. A certificate authority would not typically be a person, but would more likely be a company whose business it is to certify public keys, a governmental agency, or perhaps a department within an organization (although in this latter case the CA would likely only be used by people within the organization). Anyone who wants to rely on the services of the CA would have to obtain a legitimate copy of the CA's public key pk_{CA}. Clearly, this step must be carried out in a secure fashion since if some party obtains an incorrect version of pk_{CA} then that party may not be able to obtain an authentic copy of anyone else's public key. This means that pk_{CA} must be distributed over an *authenticated* channel. The easiest way of doing this is via physical means: for example, if the CA is within an organization then any employee can obtain an authentic copy of pk_{CA} directly from the CA on their first day of work. If the CA is a company, then other users would have to go to this company at some point and, say, pick up a copy of a CD-ROM that contains the CA's public key. The point, once again, is that this relatively inconvenient step is only carried out once.

A common way for a CA to distribute its public key in practice is to "bundle" this public key with some other software. In fact this occurs today with most popular web browsers: a CA's public key is provided together with the web browser in something called a "certificate store", and the web browser can be programmed to automatically verify certificates as they arrive, using the public key in the store. (Actually, web browsers typically have public keys of *multiple* CAs hard-wired into their code, and so more accurately fall into the "multiple CA" case discussed below.) As another example, a public key could be included as part of the operating system when a new computer is purchased.

The mechanism by which a party Bob obtains a certificate from the CA must also be very carefully controlled. As one example, Bob may have to show up in person before the CA with a CD-ROM containing his public key pk_B as well as some identification proving that his name (or his email address) is what he says it is. Only then should the CA issue an appropriate certificate for Bob's public key.

In the model where there is a single CA, parties completely trust this CA to issue certificates only when appropriate; this is why it is crucial that a detailed verification process be used before a certificate is issued. As a consequence, if Alice receives a certificate $\mathsf{cert}_{CA \to B}$ certifying that pk_B is Bob's public key, Alice will accept this assertion as valid, and use pk_B as Bob's public key.

Multiple certificate authorities. While the model in which there is only a single CA is very simple and appealing, it is not very practical. For one thing, outside of a single organization it is highly unlikely for *everyone* to trust the same CA. This need not imply that anyone thinks the CA is corrupt; it could simply be the case that someone finds the CA's verification process to be

insufficient (say, the CA asks for only one ID when generating a certificate but Alice would prefer that two IDs be used instead). Moreover, the CA is a single point of failure for the entire system. If the CA is corrupt, or can be bribed, or even if the CA is merely lax with the way it protects its private signing key, the legitimacy of issued certificates may be called into question. Reliance on a single CA is also a problem even in non-adversarial environments: if the CA is unreachable then no new certificates can be issued, and the load on the CA may be very high if many parties want to obtain certificates at once (although this is still much better than the KDC model because the CA does not need to be online once certificates are issued).

One approach to alleviating these issues is to rely on multiple CAs. A party Bob who wants to obtain a certificate on his public key can choose which CA(s) it wants to issue a certificate, and a party Alice who is presented with a certificate, or even multiple certificates issued by different CAs, can choose which CA's certificates she trusts. There is no harm in having Bob obtain a certificate from every CA (apart from some inconvenience and expense for Bob), but Alice must be more careful since the security of her communications is ultimately only as good as the least-secure CA that she trusts. That is, say Alice trusts two CAs, CA_1 and CA_2, and CA_2 is corrupted by an adversary. Then although this adversary will not be able to forge certificates issued by CA_1, it will be able to generate certificates issued by CA_2 for any identity/public key of its choice. This is actually a real problem in current web browsers. As mentioned earlier, web browsers typically come pre-configured with a number of CA public keys in their certificate store, and the default setting is for all of these CAs to be treated as equally trustworthy. Essentially any company willing to pay, however, can be included as a CA. So the list of pre-configured CAs includes some reputable, well-established companies along with other, newer companies whose trustworthiness cannot be easily established. It is left to the user to manually configure their browser settings so as to only accept signed certificates from CAs that the user trusts.

Delegation and certificate chains. Another approach which alleviates some of the burden on a single CA (but does not address the security concerns of having a single point of failure) is to use *certificate chains*. We present the idea for certificate chains of length 2, though it is easy to see that everything we say generalizes to chains of arbitrary length.

Say Charlie, acting as a CA, issues a certificate for Bob as in our original discussion. Assume further that Bob's key pk_B is a public key for a signature scheme. Bob, in turn, can issue his own certificates for other parties. For example, Bob may issue a certificate for Alice of the form

$$\mathsf{cert}_{B \to A} \stackrel{\text{def}}{=} \mathsf{Sign}_{sk_B}(\text{`Alice's key is } pk_A\text{'}).$$

Now, if Alice wants to communicate with some fourth party Dave who knows Charlie's public key (but not Bob's), then Alice can send

$$pk_A,\ \mathsf{cert}_{B \to A},\ pk_B,\ \mathsf{cert}_{C \to B},$$

to Dave. What can Dave tell from this? Well, he can first verify that Charlie, whom he trusts and whose public key is already in his possession, has signed a certificate $\mathsf{cert}_{C \to B}$ indicating that pk_B indeed belongs to someone named Bob. Dave can also verify that this person named Bob has signed a certificate $\mathsf{cert}_{B \to A}$ indicating that pk_A indeed belongs to Alice. If Dave trusts Charlie to only issue certificates to trustworthy people, then Dave may accept pk_A as being the authentic key of Alice.

In this example stronger semantics are associated with a certificate $\mathsf{cert}_{C \to B}$. In all our prior discussion, a certificate of this form was only an assertion that Bob holds the public key pk_B. Now, a certificate asserts that Bob holds the public key pk_B *and Bob should be trusted to issue other certificates.* It is not essential that all certificates issued by Charlie have these semantics, and Charlie could, for example, have two different "types" of certificates that he issues.

When Charlie signs a certificate for Bob having the stronger semantics discussed above, Charlie is *delegating* his ability to issue certificates to Bob. In effect, Bob can now act as a proxy for Charlie, issuing certificates on behalf of Charlie. Coming back to a CA-based PKI, we can imagine one "root" CA and n "second-level" CAs denoted $\mathrm{CA}_1, \ldots, \mathrm{CA}_n$. The root CA issues certificates on behalf of each of the second-level CAs, who can then in turn issue certificates for other principles holding public keys. This eases the burden on the root CA, and also makes it more convenient for parties to obtain certificates (since they may now contact the second-level CA who is closest to them, for example). On the other hand, managing these second-level CAs may be difficult, and their presence means that there are now more points of attack in the system.

The "web of trust" model. The last example of a PKI we will discuss is a fully-distributed model, with no central points of trust, called the "web of trust". A variant of this model is used by the PGP email encryption program for distribution of public keys.

In the "web of trust" model, anyone can issue certificates to anyone else and each user has to make their own decision about how much trust to place in certificates issued by other users. As an example of how this might work, say a user Alice is already in possession of public keys pk_1, pk_2, pk_3 for some users C_1, C_2, C_3. (We discuss below how these public keys might initially be obtained by Alice.) Another user Bob who wants to communicate with Alice might have certificates $\mathsf{cert}_{C_1 \to B}$, $\mathsf{cert}_{C_3 \to B}$, and $\mathsf{cert}_{C_4 \to B}$, and will send these certificates (along with his public key pk_B) to Alice. Alice cannot verify $\mathsf{cert}_{C_4 \to B}$ (since she doesn't have C_4's public key), but she can verify the other two certificates. Now she has to decide how much trust she places in C_1 and C_3. She may decide to accept pk_B if she unequivocally trusts C_1, or also if she trusts both C_1 and C_3 to a lesser extent. (She may, for example, consider it likely that either C_1 or C_3 is corrupt, but consider it unlikely for them *both* to be corrupt.)

We see that in this model, as we have described it, users are expected to collect both public keys of other parties, as well as certificates on their own public key (issued by other parties). In the context of PGP, this is often done at "key-signing parties" where PGP users get together (say, at a conference), give each other authentic copies of their public keys, and issue certificates for each other. In general the users at a key-signing party may not know each other, but they can check a driver's license, say, before accepting someone's public key.

Public keys and certificates can also be stored in a central database, and this is done for the case of PGP (see `http://pgp.mit.edu`). When Alice wants to send an encrypted message to Bob, she can search for Bob's public key in this database; along with Bob's public key, the database will return a list of all certificates it holds that have been issued for Bob's public key. It is also possible that multiple public keys for Bob will be found in the database, and each of these public keys may be certified by certificates issued by a different set of parties. Once again, Alice then needs to decide how much trust to place in any of these public keys before using them.

The web of trust model is attractive because it works at the "grass-roots" level, without requiring trust in any central authority. On the other hand, while it may work well for the average user encrypting their email, it does not seem appropriate for settings where security is more critical, or for the distribution of organizational public keys. If a user wants to communicate with his bank, for example, it is unlikely that he would trust people he met at a conference to certify his bank's public key, and also unlikely that a bank representative will go to a key-signing party to get the bank's key certified.

Invalidating Certificates

One important issue we have not yet touched upon at all is the fact that certificates should generally not be valid indefinitely. An employee may leave a company, in which case he or she is no longer allowed to receive encrypted communication from others within the company; a user's private key might also be stolen, at which point the user (assuming they know about the theft) will want to generate a new key-pair and have the old public key removed from circulation. In either of these scenarios, we need a way to render previously-issues certificates invalid.

Approaches for handling these issues are varied and complex, and we will only mention two relatively simple ideas that, in some sense, represent opposite extremes. (Improving these methods is an active area of research, and the reader is referred to the references at the end of the chapter for an introduction to the literature in this area.)

Expiration. One method for preventing certificates from being used indefinitely is to include an *expiry date* as part of the certificate. A certificate

issued by a CA Charlie for Bob's public key might now have the form

$$\mathsf{cert}_{C\to B} \stackrel{\text{def}}{=} \mathsf{Sign}_{sk_C}(\texttt{`Bob's key is } pk_B\texttt{', date}),$$

where `date` is some date in the future at which point the certificate becomes invalid. (For example, it may be 1 year from the day the certificate is issued.) When another user verifies this certificate, they need to know not only pk_B but also the expiry date, and they now need to check not only that the signature is valid, but also that the expiry date has not passed. A user who holds a certificate must contact the CA to get a new certificate issued whenever their current one expires; at this point, the CA verifies the identity/credentials of the user again before issuing another certificate.

Expiry dates provides a very coarse-grained solution to the problems mentioned earlier. If an employee leaves a company the day after getting a certificate, and the certificate expires 1 year after its issuance date, then this employee can use his or her public key illegitimately for an entire year until the expiry date passes. For this reason, this approach is typically used in conjunction with other methods such as the one we describe next.

Revocation. When an employee leaves an organization, or a user's private key is stolen, we would like the certificates that have been issued for their public keys to become invalid immediately, or at least as soon as possible. This can be achieved by having the CA explicitly *revoke* the certificate. Of course, everything we say applies more generally if the user had certificates issued by multiple CAs; for simplicity we assume a single CA.

There are many different ways revocation can be handled. One possibility (the only one we will discuss) is for the CA to include a serial number in every certificate it issues; that is, a certificate will now have the form

$$\mathsf{cert}_{C\to B} \stackrel{\text{def}}{=} \mathsf{Sign}_{sk_C}(\texttt{`Bob's key is } pk_B\texttt{', ###}),$$

where "`###`" represents the serial number of this certificate. Each certificate should have a unique serial number, and the CA will store the information $(\texttt{Bob}, pk_B, \texttt{###})$ for each certificate it generates.

If a user Bob's private key corresponding to the public key pk_B is stolen, Bob can alert the CA to this fact. (Note that the CA must verify Bob's identity here, to prevent another user from falsely revoking a certificate issued to Bob. For an alternative approach, see Exercise 12.13.) The CA will then search its database to find the serial number associated with the certificate issued for Bob and pk_B. At the end of each day, say, the CA will generate a *certificate revocation list* (or CRL) containing the serial numbers of all revoked certificates, and sign this entire list along with the current date. The signed list is then widely distributed, perhaps by posting it on the CA's public webpage.

To verify a certificate issued as above, another user now needs pk_B and also the serial number of the certificate (this can be forwarded by Bob along with everything else). Verification now requires checking that the signature

is valid, checking that the serial number does not appear on the most recent revocation list, and verifying the CA's signature on the revocation list itself.

In this approach the way we have described it, there is a gap of at most 1 day before a certificate becomes invalid. This offers more flexibility than an approach based only on expiry dates.

References and Additional Reading

Notable early work on digital signatures includes that of Diffie and Hellman [47], Rabin [118, 119], Rivest, Shamir, and Adleman [122], and Goldwasser, Micali, and Yao [71]. Lamport's one-time signature scheme was published in 1979 [92], though it was already described in [47].

Goldwasser, Micali, and Rivest [70] defined the notion of existential unforgeability under an adaptive chosen-message attack, and also gave the first construction of a stateful signature scheme satisfying this definition. (Interestingly, as explained in that paper, prior to their work some had thought that Definition 12.2 was *impossible* to achieve.) Goldreich [63] suggested an approach to make the Goldwasser-Micali-Rivest scheme stateless, and we have essentially adopted Goldreich's ideas in Section 12.6.2.

We have not shown any efficient constructions of signature schemes in this chapter since the known constructions are somewhat difficult to analyze and require cryptographic assumptions beyond those introduced in this book. The interested reader can consult, e.g., [61, 38, 55].

A tree-based construction similar in spirit to Construction 12.12 was suggested by Merkle [100, 101], though a tree-based approach was also used in [70]. Naor and Yung [106] showed that one-way permutations suffice for constructing one-time signatures that can sign messages of arbitrary length, and this was improved by Rompel [124, 84] who showed that one-way functions are sufficient. As we have seen in Section 12.6.2, one-time signatures of this sort can be used to construct signature schemes that are existentially unforgeable under an adaptive chosen-message attack.

Goldreich [65, Chapter 6] and Katz [83] provide a more extensive treatment of signature schemes than what is covered here.

The notion of certificates was first described by Kohnfelder [89] in his undergraduate thesis. Public-key infrastructures are discussed in greater detail in [87, Chapter 15] and [1]. Ellison and Schneier [51] discuss some reasons why PKI is not a panacea for identity management.

As in the private-key setting (cf. Section 4.9), one can also consider mechanisms for jointly achieving privacy and integrity in the public-key setting. This is sometimes referred to as *signcryption*; the work of An, Dodis, and Rabin [6] provides a good introduction to this area.

Exercises

12.1 Prove that the existence of a one-time signature scheme for 1-bit messages implies the existence of one-way functions.

12.2 For each of the following variants of the definition of security for signatures, state whether textbook RSA is secure and prove your answer:

(a) In this first variant, the experiment is as follows: the adversary is given the public key pk and a random message m. The adversary is then allowed to query the signing oracle once on a single message that does not equal m. Following this, the adversary outputs a signature σ and succeeds if $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$. As usual, security is said to hold if the adversary can succeed in this experiment with at most negligible probability.

(b) The second variant is as above, except that the adversary is not allowed to query the signing oracle at all.

12.3 The textbook Rabin signature scheme is the same as the textbook RSA scheme, except using the Rabin trapdoor permutation (see Section 11.2). Show that textbook Rabin signatures have the property that an adversary can actually obtain the private key using an adaptive chosen-message attack.

12.4 Another approach (besides hashed RSA) to trying to construct secure RSA signatures is to use *encoded RSA*. Here, public and private keys are as in textbook RSA; a public encoding function enc is fixed; and the signature on a message m is computed as $\sigma := [\mathsf{enc}(m)^d \bmod N]$.

(a) How is verification performed in encoded RSA?

(b) Discuss why appropriate choice of an encoding function prevents the "no-message attack" described in Section 12.3.1.

(c) Show that encoded RSA is insecure for $\mathsf{enc}(m) = 0\|m\|0^{\ell/10}$ (where $\ell = \|N\|$, $|m| = 9\ell/10 - 1$, and m is not the all-0 message).

(d) Show that encoded RSA is insecure for $\mathsf{enc}(m) = 0\|m\|0\|m$ (where $|m| = (\|N\| - 1)/2$ and m is not the all-0 message)

12.5 Show that Construction 4.5 for constructing a variable-length MAC from any fixed-length MAC can also be used (with appropriate modifications) to construct a signature scheme for arbitrary-length messages from any signature scheme for messages of fixed length $\ell(n)$. What is the minimal length that $\ell(n)$ can be? What are the advantages and disadvantages of this construction in comparison to "hash-and-sign"?

12.6 Let f be a one-way permutation (as in Definition 6.2). Consider the following signature scheme for messages in the set $\{1,\ldots,n\}$:

- To generate keys, choose random $x \leftarrow \{0,1\}^n$ and set $y := f^n(x)$. The public key is y and the private key is x.
- To sign message $i \in \{1,\ldots,n\}$, output $f^{n-i}(x)$ (where $f^0(x) \stackrel{\text{def}}{=} x$).
- To verify signature σ on message i with respect to public key y, check whether $y \stackrel{?}{=} f^i(\sigma)$.

(a) Show that the above is not a one-time signature scheme. Given a signature on a message i, for what messages j can an adversary output a forgery?

(b) Prove that no PPT adversary given a signature on i can output a forgery on any message $j > i$ except with negligible probability.

(c) Suggest how to modify the scheme so as to obtain a one-time signature scheme.

Hint: Include two values y, y' in the public key.

12.7 Consider the Lamport one-time signature scheme. Describe an adversary who obtains signatures on *two* messages of its choice and can then forge signatures on any message it likes.

12.8 The Lamport scheme uses 2ℓ values in the public key to sign messages of length ℓ. Consider the following variant: the private key consists of 2ℓ values $x_1,\ldots,x_{2\ell}$ and the public key contains the values $y_1,\ldots,y_{2\ell}$ where $y_i = f(x_i)$. A message $m \in \{0,1\}^{\ell'}$ is mapped in a one-to-one fashion to a subset $S_m \subset \{1,\ldots,2\ell\}$ of size ℓ. To sign m, the signer reveals $\{x_i\}_{i\in S_m}$. Prove that this gives a one-time signature scheme. What is the maximum message-length ℓ' that this scheme supports?

12.9 A *strong* one-time signature scheme satisfies the following (informally): given a signature σ on a message m, it is infeasible to output $(m',\sigma') \neq (m,\sigma)$ for which σ' is a valid signature on m' (note that $m = m'$ is allowed).

(a) Give a formal definition of strong one-time signatures.

(b) Assuming the existence of one-way functions, show a one-way function for which Lamport's scheme is *not* a strong one-time signature scheme.

(c) Construct a strong one-time signature scheme based on any assumption used in this book.

Hint: Use a particular one-way function in Lamport's scheme.

12.10 Let (Gen, H) be a collision-resistant hash function, where H maps strings of length $2n$ to strings of length n. Prove that the function family $(\mathsf{Gen}, \mathsf{Samp}, H)$ is one-way (cf. Definition 7.70), where Samp is the trivial algorithm that samples a random string of length $2n$.

> **Hint:** Choosing random $x \leftarrow \{0,1\}^{2n}$ and finding an inverse of $y = H^s(x)$ does not guarantee a collision. But it does yield a collision most of the time...

12.11 At the end of Section 12.6.2, we show how a pseudorandom function can be used to make Construction 12.12 stateless. Does a similar approach work for the path-based scheme described in Section 12.6.1? If so, sketch a construction and proof. If not, explain why and modify the scheme to obtain a stateless variant.

12.12 Prove Theorem 12.14.

12.13 Assume revocation of certificates is handled in the following way: when a user Bob claims that the private key corresponding to his public key pk_B has been stolen, the user sends to the CA a statement of this fact *signed with respect to* pk_B. Upon receiving such a signed message, the CA revokes the appropriate certificate.

Explain why it is not necessary for the CA to check Bob's identity in this case. In particular, explain why it is of no concern that an adversary who has stolen Bob's private key can forge signatures with respect to pk_B.

Chapter 13

Public-Key Cryptosystems in the Random Oracle Model

In the previous three chapters, we have seen constructions of public-key encryption schemes and digital signatures based on a variety of assumptions. For the most part, however, the provably-secure schemes we have discussed and analyzed are not particularly efficient. Specifically:

- In Section 10.4.3 we saw an encryption scheme that can be proven secure based on the RSA assumption (cf. Theorem 10.19), but the efficiency of this scheme does not come close to the efficiency of the textbook RSA encryption scheme described in Section 10.4.1. In fact, no secure encryption scheme based on RSA with efficiency comparable to the textbook RSA encryption scheme is currently known. (The padded RSA encryption scheme with $\ell = \Theta(n)$ is efficient, but its security is not known to follow from the assumption that the RSA problem is hard.)

- We have not shown any public-key encryption scheme that is secure against chosen-ciphertext attacks. Though efficient schemes based on the decisional Diffie-Hellman assumption and others are known, there is no known scheme based on RSA that is even remotely practical.

- We have seen only a single example of a digital signature scheme that is existentially unforgeable under an adaptive chosen-message attack. This construction, shown in Section 12.6.2, is not very practical. No signature scheme is currently known that can be proven secure based on any of the assumptions introduced in this book, and whose efficiency is comparable to the textbook RSA signature scheme.

The above hold even if we are willing to assume efficient pseudorandom functions (that could be instantiated using a block cipher such as DES or AES) and/or efficient collision-resistant hash functions (that could be instantiated using a cryptographic hash function such as SHA-1). We conclude that there are few public-key cryptosystems that are both: **(1)** efficient enough to be used in practice, yet **(2)** can be proven secure based on "standard" cryptographic assumptions (such as the RSA, factoring, or DDH assumptions).[1]

[1] An exception is the El Gamal encryption scheme that is both efficient and can be proven secure based on the DDH assumption.

This state of affairs presents a challenge to cryptographers, who continue to work at improving the efficiency of existing schemes, proposing new assumptions, and showing limitations to the best possible efficiency that can be achieved using existing assumptions. In the meanwhile, we are left in practice with the question of what schemes to use. While one might suggest to simply choose the most efficient provably-secure scheme currently available, the reality appears to be that people prefer to use *nothing* rather than use an inefficient scheme. Furthermore, in some cases existing solutions are not even remotely practical.[2]

Another possibility, of course, is to use an efficient but completely ad-hoc cryptosystem with no justification for its security other than, perhaps, the fact that the designers tried to attack the scheme and were unsuccessful. This flies in the face of everything we have said so far about the importance of the rigorous, modern approach to cryptography, and it should be clear that this is unacceptable! By using a scheme that merely appears "hard to break", we leave ourselves open to an adversary who is more clever than us and who *can* break the scheme. A better alternative must be sought.

13.1 The Random Oracle Methodology

An approach which has been hugely successful in practice, and offers a "middle ground" between a fully-rigorous proof of security on the one hand and no proof whatsoever on the other, is to introduce an idealized model in which to prove the security of cryptographic schemes. Though the idealization may not be an accurate reflection of reality, we can at least derive some measure of confidence in the soundness of a scheme's design from a proof within the idealized model. As long as the model is reasonable, such proofs are certainly better than no proofs at all.

The most popular example of this approach is the *random oracle model*, which posits the existence of a public, randomly-chosen function H that can be evaluated *only* by "querying" an oracle — which can be thought of as a "magic box" — that returns $H(x)$ when given input x. (We will discuss in the following section exactly how this is to be interpreted.) To differentiate things, the model we have been using until now (where no random oracle is present) is often called the "standard model".

No one seriously claims that a random oracle exists (although there have been suggestions that a random oracle could be implemented in practice using a trusted party). Rather, the random oracle model provides a formal *method-*

[2] Given the above discussion, it is justified to wonder why the El Gamal encryption scheme is not widely adopted in practice. Indeed, we have no good explanation for this.

ology that can be used to design and validate cryptographic schemes via the following two-step approach:

1. First, a scheme is designed and proven secure in the random oracle model. That is, we assume the world contains a random oracle, and construct and analyze a cryptographic scheme based on this assumption. Standard cryptographic assumptions (of the type we have seen until now) may be utilized in the proof of security as well.

2. When we want to implement the scheme in the real world, a random oracle is not available. Instead, the random oracle H in the scheme is *instantiated* with a cryptographic hash function $\hat{H}$ such as SHA-1, modified appropriately. That is, at each point where the scheme dictates that a party should query the oracle for the value $H(x)$, the party instead computes $\hat{H}(x)$ on its own.

The hope is that the cryptographic hash function used in the second step is "sufficiently good" at emulating a random oracle, so that the security proof given in the first step will carry over to the real-world instantiation of the scheme. A difficulty is that there is currently no theoretical justification for this hope, and in fact there exist (contrived) schemes that can be proven secure in the random oracle model but are insecure *no matter how the random oracle is instantiated* in the second step. Furthermore, as a practical matter it is not clear exactly what it means for a hash function to be "good" at emulating a random oracle, nor is it clear that this is an achievable goal. For these reasons, a proof of security for a scheme in the random oracle model should be viewed as providing evidence that the scheme has no "inherent design flaws", but should *not* be taken as a rigorous proof that any real-world instantiation of the scheme is secure. Further discussion on how to interpret proofs in the random oracle model is given in Section 13.1.2.

13.1.1 The Random Oracle Model in Detail

Before continuing, let us pin down exactly what the random oracle model entails. A good way to think about the random oracle model is as follows: The "oracle" is simply a box that takes a binary string as input and returns a binary string as output. The internal workings of the box are unknown and inscrutable. Everyone — both honest parties as well as adversaries — can interact with the box, where such interaction consists of entering a binary string x as input and receiving a binary string y as output; we refer to this as *querying the oracle on* x, and call x itself a *query* made to the oracle. Queries to the oracle are assumed to be private so that if some party queries the oracle on input x then no one else learns x, or even learns that this party queried the oracle at all. This makes sense, because calls to the oracle correspond in reality to local (private) evaluations of a cryptographic hash function.

It is guaranteed that the box is *consistent*: that is, if the box ever outputs y for a particular input x, then it always outputs the same answer y when given the same input x again. This means that we can view the box as implementing a function H; i.e., we simply define the function H in terms of the input/output characteristics of the box. For convenience, we thus speak of "querying H" rather than querying the box. No one "knows" the entire function H (except the box itself); at best, all that is known are the values of H on the strings that have been explicitly queried thus far.

We now discuss what it means to choose this function H at random. Any function H mapping n-bit inputs to $\ell(n)$-bit outputs can be viewed as a table indicating for each possible input $x \in \{0,1\}^n$ the corresponding output value $H(x) \in \{0,1\}^{\ell(n)}$. Using lexicographic order for the inputs, this means that any such function can be represented by a string of length $2^n \cdot \ell(n)$ bits, and conversely that every string of this length can be viewed as a function mapping n-bit inputs to $\ell(n)$-bit outputs. An immediate corollary is that there are exactly $U \stackrel{\text{def}}{=} 2^{\ell(n)\cdot 2^n}$ different functions having the specified input and output lengths. Picking a function H of this type uniformly at random means choosing H uniformly from among these U possibilities. In the random oracle model as we have been picturing it, this corresponds to initializing the oracle by choosing such an H and having the oracle answer according to H. Note that storing the string/table representing H in any physical device would require an *exponential* (in the input length) number of bits, so even for moderately-sized inputs this is not something we can hope to do in the real world.

An equivalent, but often more convenient, way to think about choosing a function H uniformly at random is to imagine generating random outputs for H "on-the-fly," as needed. Specifically, imagine that the function is defined by a table of pairs $\{(x_i, y_i)\}$ that is initially empty. When the oracle receives a query x it first checks whether $x = x_i$ for some pair (x_i, y_i) in the table; if so, the corresponding y_i is returned. Otherwise, a random string $y \in \{0,1\}^{\ell(n)}$ is chosen, the answer y is returned, and the oracle stores (x, y) in its table so the same output can be returned if the same input is ever queried again. While one could imagine carrying this out in the real world, this is further from our conception of "fixing" the function H once-and-for-all before beginning to run some cryptographic scheme. From the point of view of the parties interacting with the oracle, however, it is completely equivalent. We remark that viewing H as choosing output values "on-the-fly" also makes things much easier (technically as well as conceptually) when H is defined over an infinite domain such as $\{0,1\}^*$.

When we defined pseudorandom functions in Section 3.6.1, we also considered algorithms having oracle access to a random function. Lest there be any confusion, we note that the usage of a random function there is very different from the usage of a random function here. There, a random function was used *as a way of defining* what it means for a concrete keyed function to be

pseudorandom. In the random oracle model, the random function is used *as part of the construction of the primitive*, and so must somehow be instantiated in the real world if we want a concrete realization of the primitive.

Security Proofs in the Random Oracle Model

(This section may be skipped on a first reading, and should be more clear after reading the proofs in Sections 13.2 and 13.3.)

Definitions and security proofs in the random oracle model are a bit different from their counterparts in the standard model that we are, by now, familiar with. In the standard model, we have a concrete scheme Π, an experiment $\mathsf{Expt}_{\mathcal{A},\Pi}$ defined for Π and any adversary $\mathcal{A}$, and a value γ indicating the maximum desired probability of some "bad" event (e.g., for encryption $\gamma = \frac{1}{2}$ and for signatures $\gamma = 0$). A definition of security for Π then takes the following general form: the scheme Π is secure if, for any probabilistic polynomial-time adversary $\mathcal{A}$ we have

$$\Pr[\mathsf{Expt}_{\mathcal{A},\Pi}(n) = 1] \leq \gamma + \mathsf{negl}(n),$$

where *the probability is taken over the random choices of the parties running* Π *and those of the adversary* $\mathcal{A}$. Parties who use Π (in the real world) will make random choices, and so the above guarantees security in real-world usage of the scheme.

In the random oracle model, in contrast, a scheme Π will be defined in such a way that it relies on an oracle H. A concrete scheme is only obtained by fixing H and, here, we let Π^H denote the scheme that is obtained in this way. (We will not make this explicit in later sections.) A definition of security for Π then takes the following general form: Π is secure if, for any probabilistic polynomial-time adversary $\mathcal{A}$ we have

$$\Pr[\mathsf{Expt}_{\mathcal{A}^H,\Pi^H}(n) = 1] \leq \gamma + \mathsf{negl}(n).$$

(We also give $\mathcal{A}$ oracle access to H since this is the only way for $\mathcal{A}$ to evaluate H. Again, we will not make this explicit in later sections.) Now, however, *the above probability is taken over* **random choice of** H *as well as the random choices of the parties running* Π *and those of the adversary* $\mathcal{A}$. It is precisely because the probability is now also taken over choice of H that we speak of H as a *random* oracle.

To use Π in the real world, some H must be fixed. Unfortunately, security of Π is not guaranteed for any *particular* function H but is instead only guaranteed with all but negligible probability over random choice of H. (This is analogous to the fact that a scheme secure in the standard model is not guaranteed to be secure for any particular set of random choices made by the honest parties but only with high probability over these random choices.) This indicates one reason why it is difficult to argue that any concrete instantiation of the oracle H by a deterministic function like SHA-1 yields a real-world

implementation of Π that is actually secure: for all we know, SHA-1 may be one of the functions for which the scheme happens to be insecure. An additional difficulty is that once a concrete function H is fixed, the adversary $\mathcal{A}$ is no longer restricted to querying H as an oracle but can instead look at the *code* of H and use this additional information in the course of its attack.

Simple Illustrations of the Random Oracle Model

At this point some examples may be helpful. The examples given here are rather simple, do not use the full power that the random oracle model affords, and do not really illustrate any of the *limitations* of the random oracle methodology; the intention of including these examples is merely to provide a gentle introduction to the use of the model.

In all that follows, we assume a random oracle mapping n_1-bit inputs to n_2-bit outputs where $n_1, n_2 \geq n$, the security parameter. (Technically speaking, n_1 and n_2 are thus functions of n.)

A random oracle as a one-way function. We first show that a random oracle acts like a one-way function. Note that we do not say that a random oracle *is* a one-way function, since (as discussed in the previous section) a random oracle is not a fixed function. Rather, what we claim is that any polynomial-time adversary $\mathcal{A}$ succeeds with only negligible probability in the following experiment:

1. A random function H is chosen.
2. A random input $x \in \{0,1\}^{n_1}$ is chosen, and $y := H(x)$ is evaluated.
3. $\mathcal{A}$ is given y, and succeeds if it outputs a value x' such that $H(x') = y$.

We stress that $\mathcal{A}$ can query the oracle H arbitrarily many times in this experiment. (Recall that H is assumed to be accessible to everyone.) This makes sense because, in reality, the oracle is replaced by a concrete function that is known to everyone, including the adversary.

We now argue why any polynomial-time $\mathcal{A}$ succeeds in the above experiment with only negligible probability. Assume without loss of generality that $\mathcal{A}$ never makes the same query to H twice, and that the value x' output by $\mathcal{A}$ was queried by $\mathcal{A}$ to the oracle. Assume further that if $\mathcal{A}$ ever makes a query x_i with $H(x_i) = y$, then $\mathcal{A}$ succeeds. (This just means that $\mathcal{A}$ does not act stupidly and fail to output a correct answer if it finds one.) Then the success probability of $\mathcal{A}$ in the above experiment is exactly the same as the success probability of $\mathcal{A}$ in the following experiment (to see why, it helps to recall the discussion from the previous section regarding "on-the-fly" selection of a random function):

1. A random $x \in \{0,1\}^{n_1}$ is chosen, and a random value $y \in \{0,1\}^{n_2}$ is given to $\mathcal{A}$.
2. Each time $\mathcal{A}$ makes a query x_i to the random oracle, do:
   - If $x_i = x$, then $\mathcal{A}$ immediately succeeds.
   - Otherwise, choose a random $y_i \in \{0,1\}^{n_2}$. If $y_i = y$ then $\mathcal{A}$ immediately succeeds; if not, return y_i to $\mathcal{A}$ as the answer to the query and continue the experiment.

Let q be the number of queries $\mathcal{A}$ makes to the oracle, with $q = \mathsf{poly}(n)$ since $\mathcal{A}$ runs in polynomial time. Since y is completely independent of x, the probability that $\mathcal{A}$ succeeds by querying $x_i = x$ for some i is at most $q/2^{n_1}$. Furthermore, since the answer y_i is chosen at random when the query x_i is not equal to x, the probability that $\mathcal{A}$ succeeds because $y_i = y$ for some i is at most $q/2^{n_2}$. Since $n_1, n_2 \geq n$ the probability that $\mathcal{A}$ succeeds is therefore at most $2q/2^n = \mathsf{poly}(n)/2^n$, which is negligible.

A random oracle as a collision-resistant hash function. It is not much more difficult to see that a random oracle also acts like a collision-resistant hash function. That is, the success probability of any polynomial-time adversary $\mathcal{A}$ in the following game is negligible:

1. A random function H is chosen.
2. $\mathcal{A}$ succeeds if it outputs x, x' with $H(x) = H(x')$ but $x \neq x'$.

To see this, assume without loss of generality that $\mathcal{A}$ only outputs values x, x' that it had previously queried to the oracle, and that $\mathcal{A}$ never makes the same query to the oracle twice. Letting the oracle queries of $\mathcal{A}$ be $x_1, \ldots, x_q$, with $q = \mathsf{poly}(n)$, it is clear that the probability that $\mathcal{A}$ succeeds is upper-bounded by the probability that $H(x_i) = H(x_j)$ for some $i \neq j$. Viewing the choice of a random H as being computed "on-the-fly", this is exactly equal to the probability that if we pick ℓ strings $y_1, \ldots, y_\ell \in \{0,1\}^{n_2}$ independently and uniformly at random, we have $y_i = y_j$ for some $i \neq j$. The problem has now been transformed into an example of the birthday problem. Using the results of Appendix A.4 we see that $\mathcal{A}$ succeeds with probability $\mathcal{O}(q^2/2^{n_2})$, which is negligible.

Constructing a pseudorandom function from a random oracle. It is also rather easy to construct a pseudorandom function in the random oracle model (though the proof is not quite as trivial as in the examples above). Suppose $n_1 = 2n$ and $n_2 = n$. Then define

$$F_k(x) \stackrel{\text{def}}{=} H(k \| x),$$

where $|k| = |x| = n$. We claim that this is a pseudorandom function; namely, for any polynomial-time $\mathcal{A}$ the success probability of $\mathcal{A}$ in the following experiment is at most negligibly greater than $1/2$:

1. A random function H, a random $k \in \{0,1\}^n$, and a random bit b are chosen.

2. If $b = 0$, the adversary $\mathcal{A}$ is given access to an oracle for $F_k(\cdot)$. If $b = 1$, then $\mathcal{A}$ is given access to a random function mapping n-bit inputs to n-bit outputs. (This random function is *independent* of H.)

3. $\mathcal{A}$ outputs a bit b', and succeeds if $b = b'$.

In step 2, $\mathcal{A}$ can access H in addition to the function oracle provided to it by the experiment. (For this reason, H itself — with no key — is not a pseudorandom function. In other words, a pseudorandom function in the random oracle model must be indistinguishable from random *even given access to* H.) In Exercise 13.1 you are asked to show that the construction above indeed gives a pseudorandom function.

An interesting aspect of all the above proofs is that they hold even for *computationally-unbounded* adversaries, as long as such adversaries are limited to making only polynomially-many queries to the oracle. This has no real-world counterpart where, for example, *any* function can be inverted by an adversary running for an unlimited amount of time and, moreover, there is no way to define what it means to "evaluate a function" polynomially-many times (since it may be possible to determine the output of a function at multiple points without explicitly computing the function).

Advanced Proof Techniques in the Random Oracle Model

The preceding examples may not make clear that the random oracle model enables certain proof techniques that have no counterpart in the standard model, precisely due to the fact that the adversary is given only oracle access to H (and cannot evaluate H on its own). We sketch these proof techniques here, but caution the reader that a full understanding will likely have to wait until later in this chapter when these techniques are used in the proofs of some concrete schemes.

A first distinctive feature of the random oracle model, used already in the previous section, is:

> *If an adversary $\mathcal{A}$ has not* **explicitly** *queried the oracle on some point x, then the value of $H(x)$ is* **completely random** *(at least as far as $\mathcal{A}$ is concerned).*

This may seem superficially similar to the guarantee provided by a pseudorandom generator, but is actually quite different. If G is a pseudorandom generator then $G(x)$ is pseudorandom to an observer *assuming x is chosen uniformly at random and is completely unknown to the observer.* For H a random oracle, however, $H(x)$ is truly random as long as the adversary has not queried x. This is true even if x is known, or if x is not chosen uniformly

at random but *is* chosen with enough uncertainty to make guessing x difficult. (E.g., if x is an n-bit string where the first half of x is known and the last half is random then $G(x)$ might be easy to distinguish from random but $H(x)$ will not be.)

Say we are trying to prove security of some scheme in the random oracle model. As in the rest of the book, we will construct a *reduction* showing how any adversary $\mathcal{A}$ breaking the security of the scheme (in the random oracle model) can be used to violate some cryptographic assumption.[3] As part of the reduction, the random oracle with which $\mathcal{A}$ interacts must be simulated. That is: $\mathcal{A}$ will submit queries to, and receive answers from, what it believes to be the oracle, but the reduction itself must now answer these queries. This turns out to give a lot of power. For starters:

> *The reduction may choose values for the output of H as it likes (as long as these values are correctly distributed, i.e., uniformly random).*

This is sometimes called "programmability". Although it may not seem like programmability confers any advantage, it does; this is perhaps illustrated best by the proof of Theorem 13.11.

Another advantage that is derived from the fact that the reduction is able to simulate the random oracle is:

> *The reduction "sees" all the queries that $\mathcal{A}$ makes to the random oracle.*

(This does not contradict the fact, mentioned earlier, that queries to the random oracle are "private". While that is true in the formal model itself, here we are using $\mathcal{A}$ as a subroutine within a reduction.) This also turns out to be extremely useful, as the proofs of Theorems 13.2 and 13.6 will demonstrate.

There is no counterpart to either of the above once the oracle H is instantiated as a concrete function H. Once H is fixed at the outset of some experiment, the reduction can no longer set the values for the output of H as it likes, and can no longer "see" the inputs on which $\mathcal{A}$ is evaluating H. This is discussed in further detail below.

13.1.2 Is the Random Oracle Methodology Sound?

Recall that schemes proven secure in the random oracle model are implemented in the real world by instantiating H with some concrete function. With the mechanics of the random oracle model behind us, we turn to more fundamental questions such as:

[3] In contrast, the proofs of one-wayness, collision-resistance, and pseudorandomness in the previous section were information-theoretic and were not based on any cryptographic assumption.

- *What do proofs of security in the random oracle model guarantee in the real world?*
- *Are proofs in the random oracle model fundamentally different from proofs in the standard model?*

We highlight at the outset that these questions do not currently have any definitive answers: there is currently much debate within the cryptographic community regarding the role played by the random oracle model, and an active area of research is to determine what, exactly, a proof of security in the random oracle model *does* guarantee in the real world. We can only hope to give a flavor of both sides of the debate.

Objections to the random oracle model. The starting point for arguments against using random oracles is simple: as we have already noted, there is no formal or rigorous justification for believing that a proof of security for some scheme Π in the random oracle model implies anything about the security of Π in the real world (i.e., once the random oracle H has been instantiated with any particular hash function $\hat{H}$). These are more than just theoretical misgivings. A more basic issue is that *no* concrete hash function can ever act as a "true" random oracle. For example, in the random oracle model the value $H(x)$ is "completely random" if x was not explicitly queried. The counterpart would be to require that $\hat{H}(x)$ is random (or pseudorandom) if $\hat{H}$ was not explicitly evaluated on x. How are we to interpret this in the real world? For starters, it is not even clear what it means to "explicitly evaluate" $\hat{H}$: what if an adversary knows some shortcut for computing $\hat{H}$ that doesn't involve running the actual code for $\hat{H}$? Moreover, $\hat{H}(x)$ cannot possibly be random (or even pseudorandom) since once the adversary learns the description of $\hat{H}$, the value of that function on *all* inputs is immediately defined.

Limitations of the random oracle model become more clear once we examine the proof techniques introduced in Section 13.1.1. As an example, recall that one proof technique is to use the fact that a reduction algorithm can "see" the queries that an adversary $\mathcal{A}$ makes to the random oracle. But if we replace the random oracle by a particular hash function $\hat{H}$, this means that we must provide a description of $\hat{H}$ to the adversary at the beginning of the experiment. But then $\mathcal{A}$ can evaluate $\hat{H}$ on its own, without making any *explicit* queries, and so a reduction will no longer have the ability to "see" any queries made by $\mathcal{A}$. (In fact, as noted in the previous paragraph, the notion of $\mathcal{A}$ performing distinct evaluations of $\hat{H}$ may not even be true and certainly cannot be formally defined.) Likewise, the reduction algorithm can choose the outputs of H as it wishes, something that is clearly not true when a concrete function is used.

Even if we are willing to overlook the above theoretical concerns, a practical problem is that we do not currently have a very good understanding of what it means for a concrete hash function to be "sufficiently good" at instantiating a

random oracle. For concreteness, say we want to instantiate the random oracle using (some appropriate modification of) SHA-1. While for some particular scheme Π it might be reasonable to assume that Π is secure when instantiated using SHA-1, it is much less reasonable to assume that SHA-1 can take the place of the random oracle in *every* scheme designed in the random oracle model. Indeed, as we have said earlier, we *know* that SHA-1 is not a random oracle. And it is not hard to design a scheme that can be proven secure in the random oracle model, but is completely insecure when the random oracle is replaced by SHA-1. (See Exercise 13.2.)

We emphasize that an assumption of the form "SHA-1 acts like a random oracle" is significantly different from an assumption of the form "SHA-1 is collision-resistant" or "AES is a pseudorandom function." The problem lies partly with the fact that we do not have a satisfactory *definition* of what the first statement means, while we do have such definitions for the latter two statements. In particular, a random oracle is not the same as a pseudorandom function: the latter is a *keyed* function that can only be evaluated when the key is known, and is only "random-looking" when the key is *un*known. In contrast, a random oracle is an unkeyed function that can be evaluated by anyone, yet is supposed to remain "random-looking" in some ill-defined sense.

Because of this, using the random oracle model to prove security of a scheme is *qualitatively* different from, e.g., introducing a new cryptographic assumption in order to prove a scheme secure in the standard model; therefore, proofs of security in the random oracle model are less desirable and less satisfying than proofs of security in the standard model. The division of the chapters in this book can be taken as an endorsement of this preference.

Support for the random oracle model. Given all the problems with the random oracle model, why do we use it at all? More to the point: why has the random oracle model been so influential in the development of modern cryptography (especially current practical usage of cryptography), and why does it continue to be so widely used? As we will see, the random oracle model currently enables the design of substantially more efficient schemes than those we know how to construct in the standard model. As such, there are few (if any) public-key cryptosystems used today having proofs of security in the standard model, while there are numerous widely-deployed schemes having proofs of security in the random oracle model. In addition, proofs in the random oracle model are almost universally recognized as important for schemes being considered as standards. The random oracle model has increased the confidence we have in certain efficient schemes, and has played a major role in the increasing pervasiveness with which cryptographic algorithms are deployed.

The fundamental reason is the belief, with which we concur, that:

> *A proof of security in the random oracle model is significantly better than no proof at all.*

Though some disagree, we offer the following in support of this assertion:

- A proof of security for a given scheme in the random oracle model indicates that the scheme's design is "sound", in the sense that the only possible weaknesses in a real-world instantiation of the scheme are those that arise due to a weakness in the hash function used to instantiate the random oracle. Said differently, a proof in the random oracle model indicates that the only way to "break" the scheme in the real world is to "break" the hash function itself (in some way). Thus, if the hash function is "good enough" we have some confidence in the security of the scheme. Moreover, if a given instantiation of the scheme *is* successfully attacked, we can simply replace the hash function being used with a "better" one.
- Importantly, *there have been few real-world attacks on "natural" schemes proven secure in the random oracle model*. (We do not include here attacks on "contrived" schemes like that of Exercise 13.2, but remark that great care must be taken in instantiating the random oracle as indicated by the scheme in Exercise 13.3 which was once widely used.) This gives evidence to the usefulness of the random oracle model in designing practical schemes.

Nevertheless, the above ultimately represent only intuitive speculation as to the usefulness of proofs in the random oracle model, and proofs in the standard model are still, all other things being equal, preferable. Understanding exactly what proofs in the random oracle model guarantee in the real world remains an important research question facing cryptographers today.

Summary and recommendations. Using a scheme that is proven secure in the random oracle model is significantly better than using a scheme having no proof of security at all. However, when a reasonably-efficient construction having a proof of security in the standard model is known (even if it is slightly less efficient than another construction that relies on random oracles), we recommend using this instead.

Instantiating the Random Oracle

Multiple times already in this chapter, we have stated that the random oracle can be instantiated in practice using "an appropriate modification of a cryptographic hash function". In fact, matters are complicated by a number of issues including:

- Existing cryptographic hash functions almost all follow the Merkle-Damgård paradigm (cf. Section 4.6.4), and can therefore be distinguished relatively easily from a random oracle when variable-length inputs are allowed. See Exercise 13.3.
- Frequently, it is necessary for the output of the random oracle to have a certain form; e.g., the oracle should output elements of $\mathbb{Z}_N^*$ rather than bit-strings. Cryptographic hash functions, of course, output bit-strings only.

A detailed discussion of how these issues can be dealt with in practice is beyond the scope of this book; our aim is merely to alert the reader to the subtleties that arise.

13.2 Public-Key Encryption in the Random Oracle Model

In this section we present various public-key encryption schemes in the random oracle model. We present these constructions based on the RSA problem, both for convenience as well as because these constructions are most frequently instantiated using RSA in practice. We remark, however, that they can all be instantiated using an *arbitrary* trapdoor permutation (see Section 10.7.1).

13.2.1 Security Against Chosen-Plaintext Attacks

The secure public-key encryption scheme we have previously seen based on RSA (cf. Theorem 10.19) was both inefficient and difficult to prove secure; indeed, we offered no proof. In the random oracle model, things become significantly easier. Consider the following scheme, described formally in Construction 13.1. As usual for RSA-based schemes, the public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$. To encrypt a message $m \in \{0,1\}^{\ell(n)}$, the sender chooses a random $r \leftarrow \mathbb{Z}_N^*$ and sends the ciphertext

$$\langle [r^e \bmod N],\ H(r) \oplus m\rangle,$$

where H is a function, modeled as a random oracle, mapping elements of $\mathbb{Z}_N^*$ to strings of length $\ell(n)$.

CONSTRUCTION 13.1

Let GenRSA be as usual, and let $\ell(n)$ be an arbitrary polynomial. Let H be a function whose domain can be set to $\mathbb{Z}_N^*$ for any N, and whose range can be set to $\{0,1\}^{\ell(n)}$ for any n. Construct a public-key encryption scheme as follows:

- Gen: on input 1^n, run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.
- Enc: on input a public key $\langle N, e\rangle$ and a message $m \in \{0,1\}^{\ell(n)}$, choose a random $r \leftarrow \mathbb{Z}_N^*$ and output the ciphertext

$$\langle [r^e \bmod N],\ H(r) \oplus m\rangle\,.$$

- Dec: on input a private key $\langle N, d\rangle$ and a ciphertext $\langle c_1, c_2\rangle$, compute $r := [c_1^d \bmod N]$ and then output the message $H(r) \oplus c_2$.

CPA-secure RSA encryption in the random oracle model.

Assuming that the RSA problem is hard relative to GenRSA, we can argue intuitively that the scheme is CPA-secure in the random oracle model as follows: since r is chosen at random it is infeasible for an eavesdropping adversary to recover r from $c_1 = [r^e \bmod N]$. The adversary will therefore never query r to the random oracle, and so the value $H(r)$ is completely random from the adversary's point of view. But then c_2 is just a "one-time pad"-like encryption of m using the random value $H(r)$, and so the adversary gets no information about m. This intuition is developed into a formal proof below.

The proof, as indicated by the intuition above, relies heavily on the fact that H is a random oracle, and does not work if H is replaced by, e.g., a pseudorandom generator G. The reason is that the RSA assumption implies only that (roughly speaking) an adversary cannot recover r from $[r^e \bmod N]$, *but says nothing about what partial information about r the adversary might recover.* For instance, it may be the case that the adversary *can* compute half the bits of r, and in this case we can no longer claim that $G(r)$ is pseudorandom (since pseudorandomness of $G(r)$ requires r to be completely random). However, when H is a random oracle it does not matter if partial information about r is leaked; $H(r)$ is random as long as r has not been explicitly queried to the oracle.

THEOREM 13.2 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, Construction 13.1 has indistinguishable encryptions under a chosen-plaintext attack.*

PROOF Let Π denote Construction 13.1. As usual, we prove that Π has indistinguishable encryptions in the presence of an eavesdropper; by Theorem 10.10 this implies that Π is CPA-secure.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1].$$

For the reader's convenience, we describe the steps of experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. We highlight the fact that H is chosen at random as part of the experiment, as discussed previously.

1. *A random function H is chosen.*
2. $\mathsf{GenRSA}(1^n)$ *is run to generate (N, e, d). $\mathcal{A}$ is given $pk = \langle N, e\rangle$, and may query $H(\cdot)$. Eventually, $\mathcal{A}$ outputs two messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.*
3. *A random bit $b \leftarrow \{0,1\}$ and a random $r \leftarrow \mathbb{Z}_N^*$ are chosen, and $\mathcal{A}$ is given the ciphertext $\langle [r^e \bmod N], H(r) \oplus m_b\rangle$. The adversary may continue to query $H(\cdot)$.*

4. $\mathcal{A}$ then outputs a bit b'. The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

In an execution of experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$, let Query denote the event that, at any point during its execution, $\mathcal{A}$ queries r to the random oracle H (where r is the value used to generate the challenge ciphertext). We also use $\mathsf{Success}$ as shorthand for the event that $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$. Then

$$\begin{aligned}\Pr[\mathsf{Success}] &= \Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Success} \wedge \mathsf{Query}] \\ &\leq \Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Query}],\end{aligned}$$

where all probabilities are taken over the randomness used in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. We show that $\Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2}$ and that $\Pr[\mathsf{Query}]$ is negligible. The theorem follows.

CLAIM 13.3 *If H is modeled as a random oracle, then*

$$\Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2}.$$

If $\Pr[\overline{\mathsf{Query}}] = 0$ then the claim is immediate. Otherwise, we have

$$\begin{aligned}\Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] &= \Pr\left[\mathsf{Success} \mid \overline{\mathsf{Query}}\right] \cdot \Pr[\overline{\mathsf{Query}}] \\ &\leq \Pr\left[\mathsf{Success} \mid \overline{\mathsf{Query}}\right].\end{aligned}$$

Furthermore, $\Pr\left[\mathsf{Success} \mid \overline{\mathsf{Query}}\right] = \frac{1}{2}$. This is an immediate consequence of what we said earlier: namely, that if $\mathcal{A}$ does not explicitly query r to the oracle then $H(r)$ is completely random from $\mathcal{A}$'s point of view, and so $\mathcal{A}$ has no information as to whether m_0 or m_1 was encrypted. (This is exactly as in the case of the one-time pad encryption scheme.) Therefore, the probability that $b' = b$ when Query does not occur is exactly $\frac{1}{2}$. The reader should convince him or herself that this intuition can be turned into a formal proof.

CLAIM 13.4 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, then* $\Pr[\mathsf{Query}]$ *is negligible.*

The intuition here is that if Query is not negligible then we can use $\mathcal{A}$ to solve the RSA problem with non-negligible probability as follows: given inputs N, e, and $c_1 \in \mathbb{Z}_N^*$, give to $\mathcal{A}$ the public key $\langle N, e\rangle$ and ciphertext $\langle c_1, c_2\rangle$, where $c_2 \leftarrow \{0,1\}^{\ell(n)}$ is a random string. Then *monitor all the queries that $\mathcal{A}$ makes to the random oracle*. (See the discussion in Section 13.1.1.) If Query occurs then one of $\mathcal{A}$'s queries r satisfies $r^e = c_1 \bmod N$, and so we can output r as the answer. We therefore solve the RSA problem with probability exactly $\Pr[\mathsf{Query}]$, which must be negligible because the RSA problem is hard relative to GenRSA.

Formally, consider the following algorithm:

Algorithm $\mathcal{A}'$:
The algorithm is given $(N, e, \hat{c}_1)$ as input.

1. Choose a random $\hat{k} \leftarrow \{0,1\}^{\ell(n)}$.

   /* $\mathcal{A}'$ *implicitly sets* $H(\hat{r}) = \hat{k}$, *where* $\hat{r} \stackrel{\text{def}}{=} [\hat{c}_1^{1/e} \bmod N]$. *Note, however, that* $\mathcal{A}'$ *does not know* $\hat{r}$. */

2. Run $\mathcal{A}$ on input the public key $pk = \langle N, e \rangle$. Store pairs of strings $(\cdot, \cdot)$ in a table, initially empty. When $\mathcal{A}$ makes a query x to the random oracle H, answer it as follows:
   - If there is an entry (x, k) in the table, return k.
   - If $x^e = \hat{c}_1 \bmod N$, return $\hat{k}$ and store $(x, \hat{k})$ in the table. (In this case we have $x = \hat{r}$, for $\hat{r}$ defined as above.)
   - Otherwise, choose a random $k \leftarrow \{0,1\}^{\ell(n)}$, return k to $\mathcal{A}$, and store (x, k) in the table.
3. At some point, $\mathcal{A}$ outputs messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.
4. Choose a random $b \leftarrow \{0,1\}$ and set $c_2 := \hat{k} \oplus m_b$. Give $\mathcal{A}$ the ciphertext $\langle \hat{c}_1, c_2 \rangle$. Continue answering random oracle queries as before.
5. At the end of $\mathcal{A}$'s execution (after it has output its guess b'), let $x_1, \ldots, x_p$ be the list of all oracle queries made by $\mathcal{A}$. If there exists an i for which $x_i^e = \hat{c}_1 \bmod N$, output x_i.

It is immediate that $\mathcal{A}'$ runs in polynomial time. Say the input to $\mathcal{A}'$ is generated by running $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and then choosing $\hat{c}_1 \leftarrow \mathbb{Z}_N^*$ at random (see Definition 7.46). Then the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. (In each case $\langle N, e \rangle$ is generated the same way; $\hat{c}_1$ is equal to $[r^e \bmod N]$ for a randomly-chosen $r \leftarrow \mathbb{Z}_N^*$; and the random oracle queries of $\mathcal{A}$ are answered with random strings.) Thus, the probability of event Query remains unchanged. Furthermore, $\mathcal{A}'$ correctly solves the given RSA instance whenever Query occurs. That is,

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] = \Pr[\mathsf{Query}].$$

Since the RSA problem is hard relative to GenRSA, it must be the case that $\Pr[\mathsf{Query}]$ is negligible. This concludes the proof of the claim, and hence the proof of the theorem. ■

A critical aspect of the above proof is that the reduction algorithm $\mathcal{A}'$ can see the queries that $\mathcal{A}$ makes to the random oracle H. It is this property that enables $\mathcal{A}'$ to correctly solve the given RSA instance whenever $\mathcal{A}$ makes the

"right" query (i.e., whenever event Query occurs). Note that the proof also uses the fact that the reduction can implicitly set $H(\hat{r}) = \hat{k}$ without even knowing $\hat{r}$. Thus it also uses the "programmability" feature of random oracle proofs.

13.2.2 Security Against Chosen-Ciphertext Attacks

We have not yet seen any examples of public-key encryption schemes secure against chosen-ciphertext attacks. Although such schemes exist, they are somewhat complex. Moreover, no *practical* schemes are known that can be based on the RSA or factoring assumptions (in contrast to the DDH assumption for which such schemes do exist). Once again, the situation becomes much simpler in the random oracle model, and we show a construction based on the RSA assumption here.

Let GenRSA be as in the previous section, and let $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a *private-key* encryption scheme for messages of length $\ell(n)$. Consider the following public-key encryption scheme, described formally in Construction 13.5. The public and private keys, as usual, are $\langle N, e\rangle$ and $\langle N, d\rangle$, respectively. To encrypt a message $m \in \{0,1\}^{\ell(n)}$, the sender chooses a random $r \leftarrow \mathbb{Z}_N^*$ and sends the ciphertext

$$\left\langle [r^e \bmod N],\ \mathsf{Enc}'_{H(r)}(m) \right\rangle,$$

where H is a function, modeled as a random oracle, mapping elements of $\mathbb{Z}_N^*$ to strings of length n.

CONSTRUCTION 13.5

Let GenRSA be as in the previous section, let $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a private-key encryption scheme for messages of length $\ell(n)$, and let H be a function whose domain can be set to $\mathbb{Z}_N^*$ for any N, and whose range can be set to $\{0,1\}^n$ for any n. Construct a public-key encryption scheme as follows:

- Gen: on input 1^n, run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.
- Enc: on input a public key $\langle N, e\rangle$ and a message $m \in \{0,1\}^{\ell(n)}$, choose a random $r \leftarrow \mathbb{Z}_N^*$ and compute $k := H(r)$. Output the ciphertext
$$\left\langle [r^e \bmod N],\ \mathsf{Enc}'_k(m) \right\rangle.$$
- Dec: on input a private key $\langle N, d\rangle$ and a ciphertext $\langle c_1, c_2\rangle$, compute $r := [c_1^d \bmod N]$ and set $k := H(r)$. Then output $\mathsf{Dec}'_k(c_2)$.

CCA-secure RSA encryption in the random oracle model.

Construction 13.1 can be viewed as a special case of the above, using the one-time pad as the private-key encryption scheme $(\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$. In fact, it is possible to generalize Theorem 13.2, and prove that Construction 13.5 yields a CPA-secure public-key encryption scheme whenever Π' has indistinguishable encryptions in the presence of an eavesdropper (see Exercise 13.4). Here, we are interested in showing that the construction gives a CCA-secure public-key encryption scheme whenever Π' itself is CCA-secure. As shown in Section 4.8, efficient private-key encryption schemes satisfying this notion of security can be constructed relatively easily.

The intuition for the proof of CCA-security, when H is modeled as a random oracle, is roughly as in the previous section. Letting r be the value used to encrypt the challenge ciphertext presented to the adversary, we will again distinguish between the case that the adversary does not query r to the random oracle H and the case that it does. In the first case, the adversary learns nothing about the key $k = H(r)$ and so we can reduce the security of the construction to the security of the private-key encryption scheme Π'. We then argue that the second case occurs with only negligible probability if the RSA problem is hard relative to GenRSA. The proof of this is now significantly more complex than in the previous section because we must now show how it is possible to simulate decryption oracle queries of the adversary without knowing the private key. We show how it is possible for the reduction to do this by "programming" the random oracle in an appropriate way.

THEOREM 13.6 *If the RSA problem is hard relative to* GenRSA*, the private-key encryption scheme* Π' *has indistinguishable encryptions under a chosen-ciphertext attack, and* H *is modeled as a random oracle, then Construction 13.5 is a public-key encryption scheme having indistinguishable encryptions under a chosen-ciphertext attack.*

PROOF Let Π denote Construction 13.5, and let $\mathcal{A}$ be a probabilistic polynomial-time adversary. Define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n) = 1].$$

For convenience, we describe the steps of experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$:

1. *A random function* H *is chosen.*
2. $\mathsf{GenRSA}(1^n)$ *is run to obtain* (N, e, d)*.* $\mathcal{A}$ *is given* $pk = \langle N, e\rangle$*, and may query both* $H(\cdot)$ *and the decryption oracle* $\mathsf{Dec}_{\langle N,d\rangle}(\cdot)$*.*
3. *Eventually* $\mathcal{A}$ *outputs two messages* $m_0, m_1 \in \{0,1\}^{\ell(n)}$*. A random* $b \leftarrow \{0,1\}$ *and* $r \leftarrow \mathbb{Z}^*_N$ *are chosen, and* $\mathcal{A}$ *is given the challenge ciphertext* $\left\langle [r^e \bmod N], \mathsf{Enc}'_{H(r)}(m_b)\right\rangle$*.*

4. *Adversary $\mathcal{A}$ may continue to query $H(\cdot)$ and the decryption oracle, though it may not query the latter on the challenge ciphertext it was given.*
5. *$\mathcal{A}$ then outputs a bit b'. The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.*

In an execution of experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$, let Query denote the event that, at any point during its execution, $\mathcal{A}$ queries r to the random oracle H. We also use $\mathsf{Success}$ as shorthand for the event that $b' = b$. Then

$$\begin{aligned}\Pr[\mathsf{Success}] &= \Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Success} \wedge \mathsf{Query}] \\ &\leq \Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Query}],\end{aligned}$$

where all probabilities are taken over the randomness used in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. We show that there exists a negligible function negl such that

$$\Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2} + \mathsf{negl}(n),$$

and that $\Pr[\mathsf{Query}]$ is negligible. The theorem follows.

CLAIM 13.7 *If the private-key encryption scheme Π' has indistinguishable encryptions under a chosen-ciphertext attack and H is modeled as a random oracle, then there exists a negligible function negl such that*

$$\Pr\left[\mathsf{Success} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2} + \mathsf{negl}(n).$$

The proof now is much more involved than the proof of the corresponding claim in the previous section. This is in part because, as discussed in the intuition preceding this theorem, Construction 13.1 uses the perfectly-secret one-time pad as its "private-key component", whereas Construction 13.5 uses a computationally-secure private-key encryption scheme Π'.

Consider the following adversary $\mathcal{A}'$ carrying out a chosen-ciphertext attack on Π' (cf. Definition 3.30):

Adversary $\mathcal{A}'(1^n)$
$\mathcal{A}'$ has access to a decryption oracle $\mathsf{Dec}'_{\hat{k}}(\cdot)$ for some (unknown) secret key $\hat{k}$.

1. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). Choose $\hat{r} \leftarrow \mathbb{Z}^*_N$ and set $\hat{c}_1 := [\hat{r}^e \bmod N]$.
   / $\mathcal{A}'$ is implicitly setting $H(\hat{r}) = \hat{k}$ */*
2. Run $\mathcal{A}$ on input $pk := \langle N, e\rangle$. Pairs of strings $(\cdot, \cdot)$ are stored in a table, initially empty. When $\mathcal{A}$ makes a query $\langle c_1, c_2\rangle$ to its decryption oracle, $\mathcal{A}'$ answers it as follows:

- If $c_1 = \hat{c}_1$, query c_2 to the decryption oracle and return the result to $\mathcal{A}$.
- If $c_1 \neq \hat{c}_1$, compute $r := [c_1^d \bmod N]$. Then compute $k := H(r)$ using the procedure discussed below. Return the result $\mathsf{Dec}'_k(c_2)$ to $\mathcal{A}$.

When the value $H(r)$ is needed, either in response to a query by $\mathcal{A}$ to the random oracle, or in the course of answering a query by $\mathcal{A}$ to its decryption oracle, compute $H(r)$ as follows:

- If there is an entry (r, k) in the table, return k.
- Otherwise, choose a random $k \leftarrow \{0,1\}^n$, return it, and store (r, k) in the table.

3. At some point, $\mathcal{A}$ outputs $m_0, m_1 \in \{0,1\}^{\ell(n)}$. Output these same messages, and receive in return a challenge ciphertext $\hat{c}_2$. Give the challenge ciphertext $\langle \hat{c}_1, \hat{c}_2 \rangle$ to $\mathcal{A}$, and continue to answer the oracle queries of $\mathcal{A}$ as before.
4. When $\mathcal{A}$ outputs its guess b', this value is output by $\mathcal{A}'$.

It is immediate that $\mathcal{A}'$ runs in polynomial time. Furthermore, $\mathcal{A}'$ never submits its challenge ciphertext $\hat{c}_2$ to its own decryption oracle after step 3; the only way this could happen would be if $\mathcal{A}$ submitted *its* challenge ciphertext $\langle \hat{c}_1, \hat{c}_2 \rangle$ to its own decryption oracle, but this is not allowed.

Let $\Pr'[\cdot]$ refer to the probability of an event in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$, and let $\Pr[\cdot]$ refer, as before, to the probability of an event in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. Define $\mathsf{Success}$ and Query as above; that is, $\mathsf{Success}$ is the event that $b' = b$, and Query is the event that $\mathcal{A}$ queries $\hat{r}$ to the random oracle. The key observation is that the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ (in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$) is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$ until event Query occurs. This can be seen as follows:

- In each case, the public key given to $\mathcal{A}$ is clearly distributed identically.
- All random oracle queries of $\mathcal{A}$ are answered with a random string in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$, exactly as would be the case in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$.
- In experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$, decryption queries by $\mathcal{A}$ of the form $\langle c_1, c_2 \rangle$ with $c_1 \neq \hat{c}_1$ are answered exactly as in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$.
- As long as Query has not occurred, decryption queries by $\mathcal{A}$ of the form $\langle \hat{c}_1, c_2 \rangle$ are answered identically in experiments $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$ and $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. This can be seen by implicitly assigning $H(\hat{r})$ the value $\hat{k}$, which is a randomly-chosen value. (On the other hand, if $\mathcal{A}$ queries $H(\hat{r})$ — i.e., if Query occurs — then $\mathcal{A}'$ returns a random value k to $\mathcal{A}$ in response to this query, and k will likely not be equal to $\hat{k}$.)

Again, however, the important point is that if Query does not occur then the view of $\mathcal{A}$ is identical in both experiments; furthermore, $\Pr'[\mathsf{Query}] = \Pr[\mathsf{Query}]$ (this follows from the fact that the experiments are identical until Query occurs) and thus $\Pr'[\overline{\mathsf{Query}}] = \Pr[\overline{\mathsf{Query}}]$. So

$$\begin{aligned}\Pr'[\mathsf{Success}] \geq \Pr'[\mathsf{Success} \wedge \overline{\mathsf{Query}}] &= \Pr'[\mathsf{Success} \mid \overline{\mathsf{Query}}] \cdot \Pr'[\overline{\mathsf{Query}}] \\ &= \Pr[\mathsf{Success} \mid \overline{\mathsf{Query}}] \cdot \Pr[\overline{\mathsf{Query}}] \\ &= \Pr[\mathsf{Success} \wedge \overline{\mathsf{Query}}].\end{aligned}$$

(The above assumes $\Pr[\overline{\mathsf{Query}}] \neq 0$, but if this is not the case then we trivially have $\Pr'[\mathsf{Success} \wedge \overline{\mathsf{Query}}] = 0 = \Pr[\mathsf{Success} \wedge \overline{\mathsf{Query}}]$.) Because Π' is CCA-secure, there exists a negligible function negl such that

$$\Pr'[\mathsf{Success}] \leq \frac{1}{2} + \mathsf{negl}(n),$$

and thus

$$\Pr[\mathsf{Success} \wedge \overline{\mathsf{Query}}] \leq \Pr'[\mathsf{Success}] \leq \frac{1}{2} + \mathsf{negl}(n),$$

completing the proof of the claim.

CLAIM 13.8 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, then* $\Pr[\mathsf{Query}]$ *is negligible.*

Intuitively, $\Pr[\mathsf{Query}]$ is negligible for the same reason as in the proof of Theorem 13.2. Specifically, if an adversary queries $\hat{r}$ to H, where the challenge ciphertext is $\langle [\hat{r}^e \bmod N], \mathsf{Enc}'_{H(\hat{r})}(m_b)\rangle$, then the adversary has computed $\hat{r}$ from $[\hat{r}^e \bmod N]$ for a randomly-chosen value $\hat{r} \in \mathbb{Z}_N^*$. For a formal proof, however, we need to construct a *reduction algorithm* that uses such an adversary to solve the RSA problem, and difficulties arise due to the fact that the reduction algorithm must be able to answer the decryption oracle queries of $\mathcal{A}$ *without knowledge of the private (decryption) key.* Fortunately, the random oracle model enables a solution: to decrypt a ciphertext $\langle c_1, c_2\rangle$ (where no prior decryption query was made using the same initial component c_1), the reduction algorithm generates a random key k and returns the message $\mathsf{Dec}'_k(c_2)$; it then *implicitly* sets $H(r) = k$, where $r \stackrel{\text{def}}{=} [c_1^{1/e} \bmod N]$. Note that r may be unknown at this time, and the reduction algorithm cannot compute it, in general, without the factorization of N. Thus, the reduction must ensure consistency with both prior and later queries of $\mathcal{A}$ to the random oracle (in case r is ever queried to the random oracle in the future). This is relatively simple to do:

- When decrypting a ciphertext $\langle c_1, c_2\rangle$, the reduction first checks for any prior random oracle query $H(r)$ such that $c_1 = r^e \bmod N$; if found, then the previously-returned value of $H(r)$ is used to decrypt c_2.

- When answering a random oracle query $H(r)$, the reduction computes $c_1 := [r^e \bmod N]$ and checks whether any previous decryption query used c_1 as the first component of the ciphertext. If so, then the value k previously used to answer the decryption query is now returned as the value of $H(r)$.

A simple data structure handles both cases: the reduction will maintain a table storing all the random oracle queries and answers as in the proof of Theorem 13.2 (and as in the proof of the previous claim), except that now the table will contain *triples* rather than pairs. Two types of entries will appear in the table:

- The first type of entry has the form (r, c_1, k) with $c_1 = [r^e \bmod N]$. This entry means that the reduction has defined $H(r) = k$.

- The second type of entry has the form $(\star, c_1, k)$, which means that the value $r \stackrel{\text{def}}{=} [c_1^{1/e} \bmod N]$ is not yet known. (Again, the reduction is not able to compute this value without the factorization of N.) An entry of this sort indicates that the reduction is *implicitly* setting $H(r) = k$; so, when answering a decryption oracle query $\langle c_1, c_2 \rangle$ by $\mathcal{A}$, it will return the message $\mathsf{Dec}'_k(c_2)$. If $\mathcal{A}$ ever asks the random oracle query $H(r)$ then the reduction algorithm will return the correct answer k because it will check the table for any entry having $c_1 = [r^e \bmod N]$ as its second component.

We implement the above ideas as the following reduction algorithm $\mathcal{A}'$:

Algorithm $\mathcal{A}'$:
The algorithm is given $(N, e, \hat{c}_1)$ as input.

1. Choose random $\hat{k} \leftarrow \{0,1\}^n$. Triples $(\cdot, \cdot, \cdot)$ are stored in a table that initially contains only the tuple $(\star, \hat{c}_1, \hat{k})$.

   / Letting $\hat{r} \stackrel{\text{def}}{=} [\hat{c}_1^{1/e} \bmod N]$ (which is the answer $\mathcal{A}'$ is looking for), $\mathcal{A}'$ is implicitly setting $H(\hat{r}) = \hat{k}$ */*

2. Run $\mathcal{A}$ on input $pk := \langle N, e \rangle$. When $\mathcal{A}$ makes a query $\langle c_1, c_2 \rangle$ to the decryption oracle, answer it as follows:

   - If there is an entry in the table whose second component is c_1 (i.e., the entry is either of the form (r, c_1, k) with $r^e = c_1 \bmod N$, or of the form $(\star, c_1, k)$), let k be the third component of this entry. Return $\mathsf{Dec}'_k(c_2)$.
   - Otherwise, choose a random $k \leftarrow \{0,1\}^n$, return $\mathsf{Dec}'_k(c_2)$, and store $(\star, c_1, k)$ in the table.

   When $\mathcal{A}$ makes a query r to the random oracle, compute $c_1 := [r^e \bmod N]$ and answer the query as follows:

- If there is an entry of the form (r, c_1, k) in the table, return k.
- If there is an entry of the form $(\star, c_1, k)$ in the table, return k and store (r, c_1, k) in the table.
- Otherwise, choose a random $k \leftarrow \{0,1\}^n$, return k, and store (r, c_1, k) in the table.

3. At some point, $\mathcal{A}$ outputs messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$. Choose a random bit $b \leftarrow \{0,1\}$ and set $\hat{c}_2 \leftarrow \mathsf{Enc}'_{\hat{k}}(m_b)$. Give to $\mathcal{A}$ the ciphertext $\langle \hat{c}_1, \hat{c}_2 \rangle$, and continue to answer the oracle queries of $\mathcal{A}$ as before.
4. At the end of $\mathcal{A}$'s execution, if there is an entry in the table of the form $(\hat{r}, \hat{c}_1, \hat{k})$ then output $\hat{r}$.

Algorithm $\mathcal{A}'$ exactly carries out the strategy outlined earlier, with the only addition being that a random key $\hat{k}$ is chosen at the beginning of the experiment and $\mathcal{A}'$ *implicitly* sets $H([\hat{c}_1^{1/e} \bmod N]) = \hat{k}$.

It is immediate that $\mathcal{A}'$ runs in polynomial time. Say the input to $\mathcal{A}'$ is generated by running $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and then choosing $\hat{c}_1 \leftarrow \mathbb{Z}_N^*$ at random from $\mathbb{Z}_N^*$ (see Definition 7.46). Then the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. (It may take some effort to convince yourself of this, but it follows easily given the observation that there are never any inconsistencies in the oracle answers provided by $\mathcal{A}'$.) Thus, the probability of event Query remains unchanged in the two experiments. Furthermore, $\mathcal{A}'$ correctly solves the given RSA instance whenever Query occurs. That is,

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] = \Pr[\mathsf{Query}].$$

Since the RSA problem is hard relative to GenRSA, it must be the case that $\Pr[\mathsf{Query}]$ is negligible. This concludes the proof of the claim, and hence the proof of the theorem. ■

13.2.3 OAEP

The public-key encryption scheme given in Section 13.2.2 offers a fairly efficient way to achieve security against chosen-ciphertext attacks based on the RSA assumption, in the random oracle model. (Moreover, as we noted earlier, the general paradigm shown there can be instantiated using any trapdoor permutation and so can be used to construct a scheme with similar security based on the hardness of factoring.) For certain applications, though, even more efficient schemes are desirable. The main drawback of the previous scheme is that ciphertexts are longer than a single element of $\mathbb{Z}_N^*$, even when short messages are encrypted.

The *optimal asymmetric encryption padding* (OAEP) technique eliminates this drawback, and results in ciphertexts that consist of only a single element of $\mathbb{Z}_N^*$ when short messages are encrypted. (To encrypt longer messages, hybrid encryption would be used as discussed in Section 10.3.) Technically, OAEP is a padding method and not an encryption scheme, though encryption schemes that use this padding are often simply called OAEP themselves. We denote by RSA-OAEP the combination of OAEP padding with textbook RSA encryption (this will become clear from the discussion below and Construction 13.9).

OAEP is a reversible, randomized method for encoding a plaintext message m of length $n/2$ as a string $\hat{m}$ of length $2n$. (This matches Construction 13.9, but can be generalized for other message/encoding lengths.) If we let $\mathsf{OAEP}(m, r)$ denote the encoding of a message m using randomness r, then encryption using RSA-OAEP of a message $m \in \{0,1\}^{n/2}$ with respect to the public key $\langle N, e\rangle$ is carried out by choosing a random $r \leftarrow \{0,1\}^n$ and computing the ciphertext

$$[\mathsf{OAEP}(m, r)^e \bmod N].$$

(We assume $\|N\| > 2n$.) To decrypt a ciphertext c, the receiver computes $\hat{m} := [c^d \bmod N]$ and then tries to compute $(m, r) := \mathsf{OAEP}^{-1}(\hat{m})$. If no such inverse exists, then the receiver knows the ciphertext is invalid (and outputs a special error symbol $\perp$). Otherwise, a unique inverse (m, r) exists and the receiver outputs the message m. The details, which include a specification of the encoding OAEP, are given in Construction 13.9.

The above is actually somewhat of a simplification, in that certain details are omitted and other choices of the parameters are possible. The reader interested in implementing RSA-OAEP is referred to the references given in the notes at the end of this chapter.

RSA-OAEP uses two functions G and H that are modeled as independent random oracles in the analysis. Though the existence of more than one random oracle was not discussed when we introduced the random oracle model in Section 13.1.1, this is interpreted in the natural way. In fact it is quite easy to use a single random oracle $\bar{H}$ to implement two independent random oracles G, H by setting $G(x) \stackrel{\text{def}}{=} \bar{H}(0x)$ and $H(x) \stackrel{\text{def}}{=} \bar{H}(1x)$.

Using only the fact that the encoding function OAEP is a one-to-one function mapping $3n/2$ bits (i.e., an $n/2$-bit message and an n-bit random string) to $2n$ bits, we see that the probability of a random element of $\mathbb{Z}_N^*$ being in the image of OAEP is $2^{-n/2}$, which is negligible. In fact, the encoding function OAEP is designed so that, intuitively, the *only* way to find an element in the image of OAEP is to choose m and r and then explicitly compute $\mathsf{OAEP}(m, r)$. Turning this intuition into a formal proof of security for the above construction is beyond the scope of this book. We only mention that if the RSA problem is hard relative to GenRSA, and G and H are modeled as independent random oracles, then RSA-OAEP can be proven to be CCA-secure for

CONSTRUCTION 13.9

Let GenRSA be as in the previous sections, and let $G : \{0,1\}^n \to \{0,1\}^n$ and $H : \{0,1\}^n \to \{0,1\}^n$ be functions. Construct a public-key encryption scheme as follows:

- Gen: on input 1^n, run $\mathsf{GenRSA}(1^{n+1})$ to obtain (N, e, d) with $\|N\| > 2n$.[4] The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.
- Enc: on input a public key $\langle N, e\rangle$ and a message $m \in \{0,1\}^{n/2}$, first choose a random $r \leftarrow \{0,1\}^n$. Then, set $m' := m\|0^{n/2}$, compute $\hat{m}_1 := G(r) \oplus m'$, and define

$$\hat{m} := \hat{m}_1 \,\|\, (r \oplus H(\hat{m}_1)) .$$

  Interpret $\hat{m}$ as an element of $\mathbb{Z}_N^*$ in the natural way and output the ciphertext $c := [\hat{m}^e \bmod N]$.
- Dec: on input a private key $\langle N, d\rangle$ and a ciphertext c, first compute $\hat{m} := [c^d \bmod N]$ and parse $\hat{m}$ as $\hat{m}_1\|\hat{m}_2$ with $|\hat{m}_1| = |\hat{m}_2| = n$. Next compute $r := H(\hat{m}_1) \oplus \hat{m}_2$, followed by $m' := \hat{m}_1 \oplus G(r)$. If the final $n/2$ bits of m' are not $0^{n/2}$, output $\perp$. Otherwise, output the first $n/2$ bits of m'.

The RSA-OAEP encryption scheme.

certain types of public exponents e (including the common case when $e = 3$). Variants of OAEP suitable for use with arbitrary public RSA exponents or, more generally, with other trapdoor permutations, are also known; see the references at the end of this chapter.

13.3 Signatures in the Random Oracle Model

Having completed our discussion of public-key encryption in the random oracle model, we now turn our attention to a construction of the *full-domain hash* (FDH) signature scheme. Though this, too, may be instantiated with any trapdoor permutation, we once again describe a scheme, called RSA-FDH, which is based on RSA.

We have actually seen the RSA-FDH scheme previously in Section 12.3.2, where it was called *hashed RSA*. Hashed RSA was obtained by applying the textbook RSA signature scheme to a *hash* of the message, rather than the message itself. To review: in the textbook RSA signature scheme, a message $m \in \mathbb{Z}_N^*$ was signed by computing $\sigma := [m^d \bmod N]$. Textbook RSA

[4] This explains our unusual choice to run GenRSA with input 1^{n+1} rather than input 1^n.

CONSTRUCTION 13.10

Let GenRSA be as in the previous sections, and let H be a function with domain $\{0,1\}^*$ and whose range can be set to $\mathbb{Z}_N^*$ for any N. Construct a signature scheme as follows:

- Gen: on input 1^n, run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d) and set the range of H to be $\mathbb{Z}_N^*$. The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.
- Sign: on input a private key $\langle N, d\rangle$ and a message $m \in \{0,1\}^*$, compute
$$\sigma := [H(m)^d \bmod N].$$
- Vrfy: on input a public key $\langle N, e\rangle$, a message m, and a signature σ, output 1 if and only if $\sigma^e \stackrel{?}{=} H(m) \bmod N$.

The RSA-FDH signature scheme.

is completely insecure, and in particular was shown in Section 12.3.1 to be vulnerable to the following attacks:

- An adversary can choose an arbitrary σ, compute $m := [\sigma^e \bmod N]$, and output (m, σ) as a forgery.
- Given (legitimately-generated) signatures σ_1 and σ_2 on messages m_1 and m_2, respectively, an adversary can compute a valid signature $\sigma := [\sigma_1 \cdot \sigma_2 \bmod N]$ on the message $m := [m_1 \cdot m_2 \bmod N]$.

In RSA-FDH (i.e., hashed RSA), the signer *hashes* m before signing it; that is, a signature on a message m is computed as $\sigma := [H(m)^d \bmod N]$; see Construction 13.10. In Section 12.3.2 we argued informally why this modification prevents the above attacks when H is a cryptographic hash function; we can now see why the attacks do not apply if H is modeled as a random oracle.

- When H is a random oracle, then for any given σ it will be hard to find an m such that $H(m) = [\sigma^e \bmod N]$. (See the discussion in Section 13.1.1 regarding why a random oracle acts like a one-way function.)
- If σ_1 and σ_2 are signatures on messages m_1 and m_2, respectively, this means that $H(m_1) = \sigma_1^e \bmod N$ and $H(m_2) = \sigma_2^e \bmod N$. It is not likely, however, that $\sigma = [\sigma_1 \cdot \sigma_2 \bmod N]$ is a valid signature on the message $m = [m_1 \cdot m_2 \bmod N]$ since there is no reason to believe that $H(m_1 \cdot m_2) = H(m_1) \cdot H(m_2) \bmod N$. (Indeed if H is a random oracle, the latter will happen with only negligible probability.)

We stress that the above merely serves as intuition, while in fact RSA-FDH is *provably* resistent to the above attacks as a consequence of Theorem 13.11 that we will prove below. We stress also that the above informal arguments can only be proven when H is modeled as a random oracle; we do not know

how to prove anything like the above if H is "only" collision-resistant, for example.

We prove below that if the RSA problem is hard relative to GenRSA, then RSA-FDH is existentially unforgeable under an adaptive chosen-message attack, in the random oracle model. Toward intuition for this result, first consider the case of existential unforgeability under a *no-message attack*; i.e., when the adversary cannot request any signatures. Here the adversary is limited to making queries to the random oracle, and we can assume without loss of generality that if the adversary outputs a purported forgery (m, σ) then the adversary had at some point previously queried $H(m)$. Letting $y_1, \ldots, y_q$ denote the answers that the adversary received in response to its q queries to the random oracle, we see that each y_i is completely random; furthermore, forging a valid signature on some message requires computing an eth root of one of these values. It is thus not hard to see that, under the RSA assumption, the adversary outputs a valid forgery with only negligible probability (since computing eth roots of random elements is exactly the RSA problem).

More formally, starting with an adversary $\mathcal{A}$ forging a valid signature in a no-message attack we construct an algorithm $\mathcal{A}'$ solving the RSA problem. Given input (N, e, y), algorithm $\mathcal{A}'$ first runs $\mathcal{A}$ on the public key $pk = \langle N, e \rangle$. It answers the random oracle queries of $\mathcal{A}$ with random elements of $\mathbb{Z}_N^*$ except for a single query, chosen at random from among the q oracle queries of $\mathcal{A}$, that is answered with y. Say $\mathcal{A}$ outputs (m, σ) with $\sigma^e = H(m) \bmod N$ (i.e., $\mathcal{A}$ outputs a forgery). If the input to $\mathcal{A}'$ was generated by choosing y at random from $\mathbb{Z}_N^*$, then the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is identically distributed to the view of $\mathcal{A}$ when attacking the original signature scheme. Furthermore, $\mathcal{A}$ has no information regarding which oracle query was answered with y. So with probability $1/q$ it will be the case that the query $H(m)$ was the one that was answered with y, in which case $\mathcal{A}'$ solves the given instance of the RSA problem by outputting $\sigma = y^{1/e} \bmod N$. We see that if $\mathcal{A}$ succeeds with probability ε, then $\mathcal{A}'$ solves the RSA problem with probability ε/q. Since q is polynomial, we conclude that ε must be negligible if the RSA assumption holds.

Handling the case when the adversary is allowed to request signatures on messages of its choice is more difficult. The complication arises since our reduction $\mathcal{A}'$ above does not, of course, know the decryption exponent d, yet now has to compute valid signatures on messages submitted by $\mathcal{A}$ to its signing oracle. This seems impossible (and possibly even contradictory!) until we realize that $\mathcal{A}'$ *can* correctly compute a signature on a message m as long as it sets $H(m)$ equal to $[\sigma^e \bmod N]$ for a *known* value σ. If σ is chosen uniformly at random then $[\sigma^e \bmod N]$ is uniformly distributed as well, and so the random oracle is still emulated "properly" by $\mathcal{A}'$. Here we are using, in an essential way, the fact that the random oracle is "programmable".

The above intuition forms the basis for the proof of the following:

THEOREM 13.11 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, then Construction 13.10 is existentially unforgeable under an adaptive chosen-message attack.*

PROOF Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ denote Construction 13.10, and let $\mathcal{A}$ be a probabilistic polynomial-time adversary. Define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n) = 1].$$

For convenience, we describe the steps of experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n)$:

1. *A random function H is chosen.*
2. $\mathsf{GenRSA}(1^n)$ *is run to obtain* (N, e, d).
3. *The adversary $\mathcal{A}$ is given $pk = \langle N, e\rangle$, and may query $H(\cdot)$ and the signing oracle* $\mathsf{Sign}_{\langle N,d\rangle}(\cdot)$. *(When $\mathcal{A}$ requests a signature on a message m, it is given $\sigma := [H(m)^d \bmod N]$ in return.)*
4. *Eventually, $\mathcal{A}$ outputs a pair (m, σ) where $\mathcal{A}$ had not previously requested a signature on m. The output of the experiment is 1 if $\sigma^e = H(m) \bmod N$, and 0 otherwise.*

Since we have already discussed the intuition above, we jump right into the formal proof. To simplify matters, we assume without loss of generality that: **(1)** $\mathcal{A}$ never makes the same random oracle query twice; **(2)** if $\mathcal{A}$ requests a signature on a message m, then it had previously queried $H(m)$; and **(3)** if $\mathcal{A}$ outputs (m, σ) then it had previously queried $H(m)$.

Let $q = q(n)$ be a (polynomial) upper-bound on the number of random oracle queries made by $\mathcal{A}$. Consider the following algorithm $\mathcal{A}'$:

Algorithm $\mathcal{A}'$:
The algorithm is given (N, e, y^*) as input.

1. Choose $j \leftarrow \{1, \ldots, q\}$.
2. Given $\mathcal{A}$ the public key $pk = \langle N, e\rangle$. Store triples $(\cdot, \cdot, \cdot)$ in a table, initially empty. An entry (m_i, σ_i, y_i) indicates that $\mathcal{A}'$ has set $H(m_i) = y_i$, and $\sigma_i^e = y_i \bmod N$.
3. When $\mathcal{A}$ makes its ith random oracle query $H(m_i)$, answer it as follows:
   - If $i = j$, return y^*.
   - Otherwise, choose a random value $\sigma_i \leftarrow \mathbb{Z}_N^*$, compute $y_i := [\sigma_i^e \bmod N]$, return y_i as the answer to the query, and store (m_i, σ_i, y_i) in the table.

When $\mathcal{A}$ requests a signature on message m, let i be such that $m = m_i$ and answer the query as follows:[5]

- If $i \neq j$ then there is an entry (m_i, σ_i, y_i) in the table. Return σ_i.
- If $i = j$ then abort the experiment.

4. At the end of $\mathcal{A}$'s execution, it outputs (m, σ). If $m = m_j$ and $\sigma^e = y^* \bmod N$, then output σ.

It is immediate that $\mathcal{A}'$ runs in probabilistic polynomial time. Say the input to $\mathcal{A}'$ is generated by running $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d), and then choosing $y^* \leftarrow \mathbb{Z}_N^*$ uniformly at random. The index j chosen by $\mathcal{A}'$ in the first step represents a guess as to which oracle query of $\mathcal{A}$ will correspond to the eventual forgery output by $\mathcal{A}$. When this guess is correct, the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ in experiment $\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n)$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n)$. This is, in part, because each of the q random oracle queries of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is indeed answered with a random value:

- The query $H(m_j)$ is answered with y^*, which a value that was chosen uniformly at random from $\mathbb{Z}_N^*$.
- Queries $H(m_i)$ with $i \neq j$ is answered with $y_i = [\sigma_i^e \bmod N]$, where σ_i is chosen uniformly at random from $\mathbb{Z}_N^*$. Since RSA is a permutation, this means that y_i is uniformly distributed in $\mathbb{Z}_N^*$ as well.

Moreover, j is independent of the view of $\mathcal{A}$, unless $\mathcal{A}$ happens to request a signature on m_j. But in this case the guess of $\mathcal{A}'$ was wrong (since $\mathcal{A}$ cannot output a forgery on m_j once it requests a signature on m_j).

When $\mathcal{A}'$ guesses correctly and $\mathcal{A}$ outputs a forgery, then $\mathcal{A}'$ solves the given instance of the RSA problem (because $\sigma^e = y^* \bmod N$ and thus σ is the inverse of y^*, as required). Since $\mathcal{A}'$ guesses correctly with probability $1/q$, we have that

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] = \varepsilon(n)/q(n).$$

Because the RSA problem is hard relative to GenRSA, there exists a negligible function negl such that

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] \leq \mathsf{negl}(n).$$

Since q is polynomial, we conclude that $\varepsilon(n)$ is negligible as well, completing the proof. ∎

[5] Here m_i denotes the ith query made to the random oracle. Recall our assumption that if $\mathcal{A}$ requests a signature on a message, then it had previously queried the random oracle on that message.

References and Additional Reading

The first formal treatment of the random oracle model was given by Bellare and Rogaway [14], though the idea of using a "random-looking" function in cryptographic applications had been suggested previously, most notably by Fiat and Shamir [54]. Proper instantiation of a random oracle based on concrete cryptographic hash functions is discussed in [14, 15, 16, 62, 37].

The seminal negative result concerning the random oracle model is given by Canetti et al. [30], who show (contrived) schemes that are secure in the random oracle model but are demonstrably insecure for *any* concrete instantiation of the random oracle.

OAEP was introduced by Bellare and Rogaway [15] and later standardized as PKCS #1 v2.1 (available from `http://www.rsa.com/rsalabs`). The original proof of OAEP was later found to be flawed; the interested reader is referred to [26, 58, 130] for further details.

The full-domain hash signature scheme was proposed by Bellare are Rogaway in their original paper on the random oracle model [14]. Later improvements include [16, 35, 36, 62], the first of which has been standardized as part of PKCS #1 v2.1.

Exercises

13.1 Prove that the pseudorandom function construction in Section 13.1.1 is indeed secure in the random oracle model.

13.2 In this exercise we show a scheme that can be proven secure in the random oracle model, but is insecure when the random oracle is instantiated with SHA-1. (This exercise is a bit informal since SHA-1 is only defined for a fixed output length. Nevertheless, it illustrates the main idea.) Let Π be a signature scheme that is secure in the standard model. Construct a signature scheme Π_y where signing is carried out as follows: if $H(0) = y$, then output the secret key; if $H(0) \neq y$, then return a signature computed using Π.

(a) Prove that for *any* value y, the scheme Π_y is secure in the random oracle model.

(b) Show that there exists a particular y for which Π_y is not secure when the random oracle is instantiated using SHA-1.

13.3 Consider a message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ where $\mathsf{Mac}_k(m) := H(k\|m)$ for a function $H : \{0,1\}^* \to \{0,1\}^n$ (note that $k \leftarrow \{0,1\}^n$ and verification is carried out in the natural way). Show that if H is modeled as a random oracle, then Π is a secure message authentication code. Show that if H is any concrete hash function that is constructed via the Merkle-Damgård transform, then Π is *not* a secure message authentication code.

13.4 Consider Construction 13.5 instantiated with a private-key encryption scheme Π' that has indistinguishable encryptions in the presence of an eavesdropper. Prove that if the RSA problem is hard relative to GenRSA and H is modeled as a random oracle, then this gives a CPA-secure public-key encryption scheme.

13.5 Say a public-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *one-way* if any PPT adversary $\mathcal{A}$ has negligible probability of success in the following experiment:

- $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
- *A message* $m \leftarrow \{0,1\}^n$ *is chosen uniformly at random, and a ciphertext* $c \leftarrow \mathsf{Enc}_{pk}(m)$ *is computed.*
- $\mathcal{A}$ *is given* pk *and* c*, and outputs a message* m'*. We say* $\mathcal{A}$ *succeeds if* $m' = m$.

(a) Show a construction of a CPA-secure public-key encryption scheme in the random oracle model based on any one-way public-key encryption scheme.

(b) Can a public-key encryption scheme where encryption is *deterministic* be one-way? If not, give a proof; if so, show a construction based on any of the assumptions introduced in this book.

13.6 Say three users have RSA public keys $\langle N_1, 3\rangle$, $\langle N_2, 3\rangle$, and $\langle N_3, 3\rangle$ (i.e., they all use $e = 3$) with $N_1 < N_2 < N_3$. Consider the following method for sending the same message m to each of these parties: choose random $r \leftarrow \mathbb{Z}^*_{N_1}$, and send to everyone the same ciphertext

$$\langle [r^3 \bmod N_1], [r^3 \bmod N_2], [r^3 \bmod N_3], H(r) \oplus m\rangle.$$

(a) Show that this is insecure, and an adversary can recover m even when H is modeled as a random oracle.

Hint: See Section 10.4.2.

(b) Show a simple way to fix this scheme and obtain a CPA-secure scheme with a ciphertext of length $3|m| + \mathcal{O}(n)$.

(c) Show a better approach that gives a ciphertext of length $|m| + \mathcal{O}(n)$.

13.7 Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme having indistinguishable encryptions under a chosen-*plaintext* attack, and let $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a private-key encryption scheme having indistinguishable encryptions under a chosen-*ciphertext* attack. Consider the following construction of a public-key encryption scheme Π^*:

CONSTRUCTION 13.12

Let $H : \{0,1\}^n \to \{0,1\}^n$ be a function. Construct a public-key encryption scheme as follows:

- Gen^*: on input 1^n, run $\mathsf{Gen}(1^n)$ to obtain (pk, sk). Output these as the public and private keys, respectively.
- Enc^*: on input a public key pk and a message $m \in \{0,1\}^n$, choose a random $r \leftarrow \{0,1\}^n$ and output the ciphertext

$$\left\langle \mathsf{Enc}_{pk}(r),\ \mathsf{Enc}'_{H(r)}(m) \right\rangle .$$

- Dec^*: on input a private key sk and a ciphertext $\langle c_1, c_2 \rangle$, compute $r := \mathsf{Dec}_{sk}(c_1)$ and set $k := H(r)$. Then output $\mathsf{Dec}'_k(c_2)$.

Does the above construction have indistinguishable encryptions under a chosen-*ciphertext* attack, if H is modeled as a random oracle? If yes, provide a proof. If not, where does the approach used to prove Theorem 13.6 break down?

Index of Common Notation

General notation:

- If A is a randomized algorithm, then $y \leftarrow A(x)$ denotes running A on input x with a uniformly-chosen random tape and assigning the output to y. We write $A(x; r)$ to denote the deterministic computation of A on input x using random tape r
- If S is a set, then $x \leftarrow S$ denotes that x is chosen uniformly at random S
- $\wedge$ denotes Boolean conjunction (the AND operator)
- $\vee$ denotes Boolean disjunction (the OR operator)
- $\oplus$ denotes the exclusive-or (XOR) operator; this operator can be applied to single bits or entire strings (and in the latter case, the XOR is bitwise)
- $\{0,1\}^n$ is the set of all binary strings of length n
- $\{0,1\}^{\leq n}$ is the set of all binary strings of positive length at most n
- $\{0,1\}^*$ is the set of all finite strings of any positive length
- 0^n (resp., 1^n) denotes the string comprised of n zeroes (resp., n ones)
- $\|x\|$ denotes the length of the binary representation of the (positive) integer x, written with leading bit 1. Note that $\log x < \|x\| \leq \log x + 1$
- $|x|$ denotes the length of the binary string x (which may have leading 0s), or the absolute value of the real number x
- $\mathcal{O}(\cdot), \Theta(\cdot), \Omega(\cdot), \omega(\cdot)$ see Appendix A.2
- $x \| y$ and (x, y) are used interchangeably to denote concatenation of the strings x and y
- $\Pr[X]$ denotes the probability of event X
- $x := y$ means that the variable x is assigned the value y
- $\log x$ denotes the base-2 logarithm of x

Crypto-specific notation:

- n is the security parameter
- PPT stands for "probabilistic polynomial time"
- $\mathcal{A}^{\mathcal{O}(\cdot)}$ denotes the algorithm $\mathcal{A}$ with oracle access to $\mathcal{O}$
- k typically denotes a secret key (as in private-key encryption and MACs)
- (pk, sk) denotes the public/private key-pair (for public-key encryption and digital signatures)
- negl denotes a negligible function; that is, a function for which for every polynomial $p(\cdot)$ there exists an integer N such that for $n > N$ it holds that $\mu(n) < 1/p(n)$.
- $\mathsf{poly}(n)$ denotes an arbitrary polynomial
- $\mathsf{polylog}(n)$ denotes $\mathsf{poly}(\log(n))$
- Func_n denotes the set of functions mapping n-bit strings to n-bit strings
- IV denotes an initialization vector (used for block cipher modes of operation and collision-resistant hash functions)

Algorithms and procedures:

- G denotes a pseudorandom generator
- F denotes a keyed function that is typically a pseudorandom function or permutation
- $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ denote the key generation, encryption, and decryption procedures, respectively, for both private- and public-key encryption. For the case of private-key encryption, when Gen is unspecified it is assumed that $\mathsf{Gen}(1^n)$ outputs $k \leftarrow \{0,1\}^n$ chosen uniformly at random.
- $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ denote the key generation, tag generation, and verification procedures, respectively, for a message authentication code. When Gen is unspecified it is assumed that $\mathsf{Gen}(1^n)$ outputs $k \leftarrow \{0,1\}^n$ chosen uniformly at random.
- $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ denote the key-generation, signature generation, and verification procedures, respectively, for a digital signature scheme.
- $\mathsf{GenPrime}$ denotes a probabilistic polynomial-time algorithm that, on input 1^n, outputs an n-bit prime except with probability negligible in n.

- $\mathsf{GenModulus}$ denotes a probabilistic polynomial-time algorithm that, on input 1^n, outputs outputs (N, p, q) where $N = pq$ and (except with negligible probability) p and q are n-bit primes.
- GenRSA denotes a probabilistic polynomial-time algorithm that, on input 1^n, outputs a modulus N, an integer $e > 0$ with $\gcd(e, \phi(N)) = 1$, and an integer d satisfying $ed = 1 \bmod \phi(N)$. Except with negligible probability, N is a product of two n-bit primes.
- $\mathcal{G}$ denotes a probabilistic polynomial-time algorithm that, on input 1^n, outputs (except with negligible probability) a description of a cyclic group $\mathbb{G}$ (where the group operation in $\mathbb{G}$ can be computed in time $\mathsf{poly}(n)$), the group order q (with $\|q\| = n$), and a generator $g \in \mathbb{G}$.

Number theory:

- $\mathbb{Z}$ denotes the set of integers
- $a \mid b$ means a divides b
- $a \nmid b$ means that a does not divide b
- $\gcd(a, b)$ denotes the greatest common divisor of a and b
- $[a \bmod b]$ denotes the remainder of a when divided by b. Note that $0 \leq [a \bmod b] < b$.
- $x_1 = x_2 = \cdots = x_n \bmod N$ means that $x_1, \ldots, x_n$ are all congruent modulo N

  Note: $x = y \bmod N$ means that a and b are congruent modulo N, whereas $x = [y \bmod N]$ means that x is equal to the remainder of y when divided by N.
- $\mathbb{Z}_N$ denotes the additive group of integers modulo N and sometimes the set $\{0, \ldots, N-1\}$
- $\mathbb{Z}_N^*$ denotes the multiplicative group of invertible integers modulo N (i.e., those that are relatively prime to N)
- $\phi(N)$ denotes the size of $\mathbb{Z}_N^*$
- $\mathbb{G}$ and $\mathbb{H}$ denote groups
- $\mathbb{G}_1 \simeq \mathbb{G}_2$ means that groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are isomorphic. If this isomorphism is given by f and $f(x_1) = x_2$ then we write $x_1 \leftrightarrow x_2$
- g is typically a generator of a group
- $\log_g h$ denotes the *discrete* logarithm of h to the base g

- $\langle g \rangle$ denotes the group generated by g
- p and q usually denote primes
- N typically denotes the product of two distinct primes p and q of equal length
- $\mathcal{QR}_p$ is the set of quadratic residues modulo p
- $\mathcal{QNR}_p$ is the set of quadratic non-residues modulo p
- $\mathcal{J}_p(x)$ is the Jacobi symbol of x modulo p
- $\mathcal{J}_N^{+1}$ is the set of elements with Jacobi symbol $+1$ modulo N
- $\mathcal{J}_N^{-1}$ is the set of elements with Jacobi symbol -1 modulo N
- $\mathcal{QNR}_N^{+1}$ is the set of quadratic non-quadratic residues modulo N having Jacobi symbol $+1$

Appendix A

Mathematical Background

A.1 Identities and Inequalities

We list some standard identities and inequalities that are used at various times throughout the text.

THEOREM A.1 (Binomial Expansion Theorem) *Let x, y be real numbers, and let n be a positive integer. Then*

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i \, y^{n-i}.$$

PROPOSITION A.2 *For all $x \geq 1$ it holds that $(1-1/x)^x \leq e^{-1}$.*

PROPOSITION A.3 *For all x it holds that $1 - x \leq e^{-x}$.*

PROPOSITION A.4 *For all x with $0 \leq x \leq 1$ it holds that*

$$e^{-x} \leq 1 - \left(1 - \frac{1}{e}\right) \cdot x \leq 1 - \frac{x}{2}\,.$$

A.2 Asymptotic Notation

We follow the standard notation for expressing the asymptotic behavior of functions.

DEFINITION A.5 *Let $f(n), g(n)$ be functions from non-negative integers to non-negative reals. Then:*

- *$f(n) = \mathcal{O}(g(n))$ means that there exist positive integers c and n' such that for all $n > n'$ it holds that $f(n) \leq c \cdot g(n)$.*

- $f(n) = \Omega(g(n))$ *means that there exist positive integers* c *and* n' *such that for all* $n > n'$ *it holds that* $f(n) \geq c \cdot g(n)$.
- $f(n) = \Theta(g(n))$ *means that* $f(n) = \mathcal{O}(g(n))$ *and* $f(n) = \Omega(g(n))$.
- $f(n) = o(g(n))$ *means that* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) = \omega(g(n))$ *means that* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Example A.6
Let $f(n) = n^4 + 3n + 500$. Then:

- $f(n) = \mathcal{O}(n^4)$.
- $f(n) = \mathcal{O}(n^5)$. In fact, $f(n) = o(n^5)$.
- $f(n) = \Omega(n^3 \log n)$. In fact, $f(n) = \omega(n^3 \log n)$.
- $f(n) = \Theta(n^4)$.

♢

A.3 Basic Probability

We assume the reader is familiar with basic probability theory, on the level of what is covered in a typical undergraduate course on discrete mathematics. Here we simply remind the reader of some notation and basic facts.

If E is an event, then $\bar{E}$ denotes the complement of that event; i.e., $\bar{E}$ is the event that E does *not* occur. By definition, $\Pr[E] = 1 - \Pr[\bar{E}]$. If E_1 and E_2 are events, then $E_1 \wedge E_2$ denotes their conjunction; i.e., $E_1 \wedge E_2$ is the event that *both* E_1 and E_2 occur. By definition, $\Pr[E_1 \wedge E_2] \leq \Pr[E_1]$. Events E_1 and E_2 are said to be *independent* if $\Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2]$.

If E_1 and E_2 are events, then $E_1 \vee E_2$ denotes the disjunction of E_1 and E_2; that is, $E_1 \vee E_2$ is the event that *either* E_1 *or* E_2 occur. It follows from the definition that $\Pr[E_1 \vee E_2] \geq \Pr[E_1]$. The *union bound* is often a very useful upper-bound of this quantity.

PROPOSITION A.7 (Union Bound)

$$\Pr[E_1 \vee E_2] \leq \Pr[E_1] + \Pr[E_2].$$

Repeated application of the union bound for any events $E_1, \ldots, E_k$ gives

$$\Pr\left[\bigvee_{i=1}^{k} E_i\right] \leq \sum_{i=1}^{k} \Pr[E_i].$$

Let $F, E_1, \ldots, E_n$ be events such that $\Pr[E_1 \vee \cdots \vee E_n] = 1$ and $\Pr[E_i \wedge E_j] = 0$ for all $i \neq j$. That is, the E_i *partition* the space of all possible events, so that with probability 1 exactly one of the events E_i occurs. Then

$$\Pr[F] = \sum_{i=1}^{n} \Pr[F \wedge E_i].$$

A special case is when $n = 2$ and $E_2 = \bar{E}_1$, giving

$$\Pr[F] = \Pr[F \wedge E_1] + \Pr[F \wedge \bar{E}_1].$$

The *conditional probability of* E_1 *given* E_2, denoted $\Pr[E_1 \mid E_2]$, is defined as

$$\Pr[E_1 \mid E_2] \stackrel{\text{def}}{=} \frac{\Pr[E_1 \wedge E_2]}{\Pr[E_2]}$$

as long as $\Pr[E_2] \neq 0$. (If $\Pr[E_2] = 0$ then $\Pr[E_1 \mid E_2]$ is undefined.) This represents the probability that event E_1 occurs once it is given that event E_2 has occurred. It follows immediately from the definition that

$$\Pr[E_1 \wedge E_2] = \Pr[E_1 \mid E_2] \cdot \Pr[E_2]\,;$$

equality holds even if $\Pr[E_2] = 0$ as long as we interpret multiplication by zero on the right-hand side in the obvious way.

We can now easily derive Bayes' theorem.

THEOREM A.8 (Bayes' Theorem) *If* $\Pr[E_2] \neq 0$ *then*

$$\Pr[E_1 \mid E_2] = \frac{\Pr[E_1] \cdot \Pr[E_2 \mid E_1]}{\Pr[E_2]}.$$

PROOF We have

$$\begin{aligned}
\Pr[E_1 \mid E_2] &= \frac{\Pr[E_1 \wedge E_2]}{\Pr[E_2]} \\
&= \frac{\Pr[E_2 \wedge E_1]}{\Pr[E_2]} \\
&= \frac{\Pr[E_2 \mid E_1] \cdot \Pr[E_1]}{\Pr[E_2]}.
\end{aligned}$$

■

A.4 The "Birthday" Problem

If we choose q elements $y_1, \ldots, y_q$ uniformly at random from a set of size N, what is the probability that there exist distinct i, j with $y_i = y_j$? We refer to the stated event as a *collision*, and denote the probability of this event by $\mathsf{coll}(q, N)$. This problem is related to the so-called *birthday problem* which asks what size group of people do we need to take such that with probability $1/2$ some pair of people in the group share a birthday? To see the relationship, let y_i denote the birthday of the ith person in the group. If there are q people in the group then we have q values $y_1, \ldots, y_q$ chosen uniformly from $\{1, \ldots, 365\}$, making the simplifying assumption that birthdays are uniformly and independently distributed among the 365 days of a non-leap year. Furthermore, matching birthdays correspond to a collision, i.e., distinct i, j with $y_i = y_j$. So the desired solution to the birthday problem is given by the minimal (integer) value of q for which $\mathsf{coll}(q, 365) \geq 1/2$. (The answer may surprise you — taking $q = 23$ people suffices!)

In this section, we prove coarse lower and upper bounds on $\mathsf{coll}(q, N)$. Taken together and summarized at a high level, they show that if $q < \sqrt{N}$ then the probability of a collision is $\Theta(q^2/N)$; alternately, for $q = \Theta(\sqrt{N})$ the probability of a collision is constant.

An upper bound for the collision probability is easy to obtain.

LEMMA A.9 *Fix a positive integer N, and say q elements $y_1, \ldots, y_q$ are chosen uniformly and independently at random from a set of size N. Then the probability that there exist distinct i, j with $y_i = y_j$ is at most $\frac{q^2}{2N}$. That is,*

$$\mathsf{coll}(q, N) \leq \frac{q^2}{2N}.$$

PROOF The proof is a simple application of the union bound (Proposition A.7). Recall that a *collision* means that there exist distinct i, j with $y_i = y_j$. Let Coll denote the event of a collision, and let $\mathsf{Coll}_{i,j}$ denote the event that $y_i = y_j$. It is immediate that $\Pr[\mathsf{Coll}_{i,j}] = 1/N$ for any distinct i, j. Furthermore, $\mathsf{Coll} = \bigvee_{i \neq j} \mathsf{Coll}_{i,j}$ and so repeated application of the union bound implies that

$$\begin{aligned} \Pr[\mathsf{Coll}] &= \Pr\left[\bigvee_{i \neq j} \mathsf{Coll}_{i,j}\right] \\ &\leq \sum_{i \neq j} \Pr[\mathsf{Coll}_{i,j}] = \binom{q}{2} \cdot \frac{1}{N} \leq \frac{q^2}{2N}. \end{aligned}$$

■

LEMMA A.10 *Fix a positive integer N, and say $q \leq \sqrt{2N}$ elements $y_1, \ldots, y_q$ are chosen uniformly and independently at random from a set of size N. Then the probability that there exist distinct i, j with $y_i = y_j$ is at least $\frac{q(q-1)}{4N}$. That is,*

$$\mathsf{coll}(q, N) \geq \frac{q(q-1)}{4N}.$$

PROOF Recall that a *collision* means that there exist distinct i, j with $y_i = y_j$. Let Coll denote this event. Let NoColl_i be the event that there is *no* collision among $y_1, \ldots, y_i$; that is, $y_j \neq y_k$ for all $j < k \leq i$. Then $\mathsf{NoColl}_q = \overline{\mathsf{Coll}}$ is the event that there is no collision at all.

If NoColl_q occurs then NoColl_i must also have occurred for all $i \leq q$. Thus,

$$\Pr[\mathsf{NoColl}_q] = \Pr[\mathsf{NoColl}_1] \cdot \Pr[\mathsf{NoColl}_2 \mid \mathsf{NoColl}_1] \cdots \Pr[\mathsf{NoColl}_q \mid \mathsf{NoColl}_{q-1}].$$

Now, $\Pr[\mathsf{NoColl}_1] = 1$ since y_1 cannot collide with itself. Furthermore, if event NoColl_i occurs then $\{y_1, \ldots, y_i\}$ contains i distinct values; so, the probability that y_{i+1} collides with one of these values is $\frac{i}{N}$ and hence the probability that y_{i+1} does *not* collide with any of these values is $1 - \frac{i}{N}$. This means

$$\Pr[\mathsf{NoColl}_{i+1} \mid \mathsf{NoColl}_i] = 1 - \frac{i}{N},$$

and so

$$\Pr[\mathsf{NoColl}_q] = \prod_{i=1}^{q-1} \left(1 - \frac{i}{N}\right).$$

Since $i/N < 1$ for all i, we have $1 - \frac{i}{N} \leq e^{-i/N}$ (by Inequality A.3) and so:

$$\Pr[\mathsf{NoColl}_q] \leq \prod_{i=1}^{q-1} e^{-i/N} = e^{-\sum_{i=1}^{q-1}(i/N)} = e^{-q(q-1)/2N}.$$

We conclude that

$$\Pr[\mathsf{Coll}] = 1 - \Pr[\mathsf{NoColl}_q] \geq 1 - e^{-q(q-1)/2N} \geq \frac{q(q-1)}{4N},$$

using Inequality A.4 in the last step (note that $q(q-1)/2N < 1$). ■

Appendix B

Supplementary Algorithmic Number Theory

For the cryptographic constructions given in this book to be efficient (i.e., to run in time polynomial in the lengths of their inputs), it is necessary for these constructions to utilize efficient (that is, polynomial-time) algorithms for performing basic number-theoretic operations. Though in some cases these number-theoretic algorithms are trivial, it is still worthwhile to pause and consider their efficiency since for cryptographic applications it is not uncommon to use integers that are *thousands* of bits long. In other cases the algorithms themselves are quite clever, and an analysis of their performance may rely on non-trivial group-theoretic results.

In Appendix B.1 we describe basic algorithms for integer arithmetic. Here we cover the familiar algorithms for addition, subtraction, etc. as well as the Euclidean algorithm for computing greatest common divisors. We also discuss the extended Euclidean algorithm, assuming for that discussion that the reader has covered the material in Section 7.1.1.

In Appendix B.2 we show various algorithms for modular arithmetic. In addition to a brief discussion of basic modular operations (i.e., modular reduction, addition, multiplication, and inversion), we discuss algorithms for problems that are less commonly encountered outside the field of cryptography: exponentiation modulo N (as well as in arbitrary groups) and choosing a random element of $\mathbb{Z}_N$ or $\mathbb{Z}_N^*$ (or in an arbitrary group). This section assumes familiarity with the basic group theory covered in Section 7.1.

The material listed above is used implicitly throughout the second half of the book, though it is not absolutely necessary to read this material in order to follow the book. (In particular, the reader willing to accept the results of this Appendix without proof can simply read the summary of these results in the theorems below.) Appendix B.3, which discusses the issue of finding generators in cyclic groups and assumes the results of Section 7.3.1, contains material that is hardly used at all; it is included for completeness and reference.

Since our goal is only to establish that certain problems can be solved in polynomial time, we have opted for *simplicity* rather than *efficiency* in our selection of algorithms and their descriptions (as long as the algorithms run in polynomial time). For this reason, we generally will not be interested in the exact running time of the algorithms we present beyond establishing that

they indeed run in polynomial time. The reader who is seriously interested in implementing these algorithms is forewarned to look at other sources for more efficient alternatives as well as various techniques for speeding up the necessary computations.

The results in this Appendix are summarized by the theorems that follow. Throughout, we assume that any integer a provided as input is written using exactly $\|a\|$ bits; i.e., the high-order bit is 1. In Appendix B.1 we show:

THEOREM B.1 (Integer Operations) *Given integers a and b, it is possible to perform the following operations in time polynomial in $\|a\|$ and $\|b\|$:*

1. *Compute the sum $a + b$ and the difference $a - b$;*
2. *Compute the product ab*
3. *Compute positive integers q and $r < b$ such that $a = qb + r$ (i.e., Compute* division with remainder*);*
4. *Compute the greatest common divisor of a and b, $\gcd(a, b)$;*
5. *Compute integers X, Y with $Xa + Yb = \gcd(a, b)$.*

The following results are proved in Appendix B.2:

THEOREM B.2 (Modular Operations) *Given integers $N > 1$, a, and b, it is possible to perform the following operations in time polynomial in $\|a\|$, $\|b\|$, and $\|N\|$:*

1. *Compute the modular reduction $[a \bmod N]$;*
2. *Compute the sum $[(a + b) \bmod N]$, the difference $[(a - b) \bmod N]$, and the product $[ab \bmod N]$;*
3. *Determine whether a is invertible modulo N;*
4. *Compute the multiplicative inverse $[a^{-1} \bmod N]$, assuming a is invertible modulo N;*
5. *Compute the exponentiation $[a^b \bmod N]$;*

The following generalizes Theorem B.2(4) to arbitrary groups:

THEOREM B.3 (Group Exponentiation) *Let $\mathbb{G}$ be a group, written multiplicatively. Let g be an element of the group and let b be a non-negative integer. Then g^b can be computed using* poly($\|b\|$) *group operations.*

THEOREM B.4 (Choosing Random Elements) *There exists a randomized algorithm with the following properties: on input N,*

- *The algorithm runs in time polynomial in $\|N\|$;*
- *The algorithm outputs* fail *with probability negligible in $\|N\|$; and*
- *Conditioned on not outputting* fail*, the algorithm outputs a uniformly-distributed element of $\mathbb{Z}_N$.*

An algorithm with analogous properties exists for $\mathbb{Z}_N^*$ as well.

By way of notation, we let $x \leftarrow \mathbb{Z}_N$ denote the random selection of an element x from $\mathbb{Z}_N$ using, e.g., the algorithm guaranteed by the above theorem (with analogous notation for $\mathbb{Z}_N^*$). Since the probability of outputting fail is negligible, we ignore it (and instead leave this possibility implicit). In Appendix B.2 we also discuss generalizations of the above to the case of selecting a random element from any finite group.

A proof of the following is in Appendix B.3:

THEOREM B.5 (Testing and Finding Generators) *Let $\mathbb{G}$ be a cyclic group of order q, and assume that the group operation and the selection of a random group element can be carried out in unit time.*

1. *There exists an algorithm that on input q, the prime factorization of q, and an element $g \in \mathbb{G}$, runs in $\mathsf{poly}(\|q\|)$ time and correctly determines whether or not g is a generator of $\mathbb{G}$.*
2. *There exists a randomized algorithm that on input q and the prime factorization of q, runs in $\mathsf{poly}(\|q\|)$ time and outputs a generator of $\mathbb{G}$ except with probability negligible in $\|q\|$. Conditioned on the output being a generator it is uniformly-distributed among the generators of $\mathbb{G}$.*

B.1 Integer Arithmetic

B.1.1 Basic Operations

We begin our exploration of algorithmic number theory with a discussion of integer addition/subtraction, multiplication, and division with remainder. A little thought shows that all these operations can be carried out in time polynomial in the input length using the standard "grade-school" algorithms for these problems. For example, addition of two positive integers a and b with $a > b$ can be done in time linear in $\|a\|$ by stepping one-by-one through the bits of a and b, starting with the low-order bits, and computing the corresponding output bit and a "carry bit" at each step. (Details are omitted.)

Multiplication of two n-bit integers a and b, to take another example, can be done by first generating a list of n integers of length at most $2n$ (each of which is equal to $a \cdot 2^{i-1} \cdot b_i$, where b_i is the ith bit of b) and then adding these n integers together to obtain the final result.

Although these grade-school algorithms already suffice to demonstrate that the aforementioned problems can be solved in polynomial time, it is interesting to note that these algorithms are in some cases not the best ones available. As an example, the simple algorithm for multiplication given above runs in time $\mathcal{O}(n^2)$ to multiply two n-bit integers, but there exists an alternate algorithm running in time $\mathcal{O}(n^{\log_2 3})$ (and even that is not the best possible). While the difference is insignificant for numbers of the size we encounter daily, it becomes noticeable when the numbers are large. In cryptographic applications it is not uncommon to use integers that are thousands of bits long (i.e., $n > 1000$), and a judicious choice of which algorithms to use then becomes critical.

B.1.2 The Euclidean and Extended Euclidean Algorithms

Recall from Section 7.1 that $\gcd(a, b)$, the *greatest common divisor* of two integers a and b, is the largest integer d that divides both a and b. We state an easy proposition regarding the greatest common divisor, and then show how this leads to an efficient algorithm for computing gcd's.

PROPOSITION B.6 *Let $a, b > 1$ with $b \nmid a$. Then*

$$\gcd(a, b) = \gcd(b, [a \bmod b]).$$

PROOF If $b > a$ the stated claim is immediate. So assume $a > b$. Write $a = qb + r$ for q, r positive integers and $r < b$ (cf. Proposition 7.1); note that $r > 0$ because $b \nmid a$. Since $r = [a \bmod b]$, we prove the proposition by showing that $\gcd(a, b) = \gcd(b, r)$.

Let $d = \gcd(a, b)$. Then d divides both a and b, and so d also divides $r = a - qb$. By definition of the greatest common divisor, we thus have $\gcd(b, r) \geq d = \gcd(a, b)$.

Let $d' = \gcd(b, r)$. Then d' divides both b and r, and so d' also divides $a = qb + r$. By definition of the greatest common divisor, we thus have $\gcd(a, b) \geq d' = \gcd(b, r)$.

Since $d \geq d'$ and $d' \geq d$, we conclude that $d = d'$. ■

The above proposition suggests the recursive *Euclidean algorithm* (Algorithm B.7) for computing the greatest common divisor $\gcd(a, b)$ of two integers a and b.

Correctness of the algorithm follows readily from Proposition B.6. As for its running time, we show below that on input (a, b) the algorithm makes fewer than $2 \cdot \|b\|$ recursive calls. Since checking whether b divides a and computing

ALGORITHM B.7
The Euclidean algorithm GCD

Input: Integers a, b with $a \geq b > 0$
Output: The greatest common divisor of a and b

if b divides a
  return b
else return $\mathsf{GCD}(b, [a \bmod b])$

$[a \bmod b]$ can both be done in time polynomial in $\|a\|$ and $\|b\|$, this implies that the entire algorithm runs in polynomial time.

PROPOSITION B.8 *Consider an execution of* $\mathsf{GCD}(a_0, b_0)$*, and let* a_i, b_i *(for* $i = 1, \ldots, \ell$*) denote the arguments to the* i*th recursive call of* GCD*. Then* $b_{i+2} \leq b_i/2$ *for* $0 \leq i \leq \ell - 2$.

PROOF Since $b_{i+1} = [a_i \bmod b_i]$, we always have $b_{i+1} < b_i$ and so the $\{b_i\}$ decrease as i increases. Fixing arbitrary i with $0 \leq i \leq \ell - 2$, we show that $b_{i+2} \leq b_i/2$. If $b_{i+1} \leq b_i/2$ we are done (because $b_{i+2} < b_{i+1}$). Otherwise, we have $b_{i+1} > b_i/2$. Now, since $a_{i+1} = b_i$, the $(i+1)$st recursive call is

$$\mathsf{GCD}(a_{i+1}, b_{i+1}) = \mathsf{GCD}(b_i, b_{i+1})$$

and

$$b_{i+2} = [a_{i+1} \bmod b_{i+1}] = [b_i \bmod b_{i+1}] = b_i - b_{i+1} < b_i/2,$$

using the fact that $b_i > b_{i+1} > b_i/2$ for both the third equality and the final inequality. ■

COROLLARY B.9 *In an execution of algorithm* $\mathsf{GCD}(a, b)$*, there are at most* $2\,\|b\| - 2$ *recursive calls to* GCD.

PROOF Let a_i, b_i (for $i = 1, \ldots, \ell$) denote the arguments to the ith recursive call of GCD. The $\{b_i\}$ are always greater than zero, and the algorithm makes no further recursive calls if it ever happens that $b_i = 1$ (since then $b_i \mid a_i$). The previous proposition indicates that the $\{b_i\}$ decrease by a multiplicative factor of (at least) 2 in every two iterations. It follows that the number of recursive calls to GCD is at most $2 \cdot (\|b\| - 1)$. ■

The Extended Euclidean Algorithm

By Proposition 7.2, we know that for positive integers a, b there exist integers X, Y with $Xa + Yb = \gcd(a, b)$. A simple modification of the Euclidean

ALGORITHM B.10
The extended Euclidean algorithm eGCD

Input: Integers a, b with $a \geq b > 0$
Output: (d, X, Y) with $d = \gcd(a, b)$ and $Xa + Yb = d$

if b divides a
  return $(b, 0, 1)$
else
  Compute integers q, r with $a = qb + r$ and $0 \leq r < b$
  $(d, X, Y) := \mathsf{eGCD}(b, r)$ /* note that $Xb + Yr = d$ */
  return $(d, Y, X - Yq)$

algorithm, called the *extended Euclidean algorithm*, can be used to find X, Y in addition to computing $\gcd(a, b)$; see Algorithm B.10 above. You are asked to show correctness of the extended Euclidean algorithm in Exercise B.1, and to prove that the algorithm runs in polynomial time in Exercise B.2.

B.2 Modular Arithmetic

We now turn our attention to some basic arithmetic operations modulo N. In this section, $N > 1$ is arbitrary unless stated otherwise. We will use $\mathbb{Z}_N$ to refer both to the set $\{0, \ldots, N-1\}$ as well as to the group that results by considering addition modulo N among the elements of this set. We use $\mathbb{Z}_N^*$ similarly.

B.2.1 Basic Operations

Efficient algorithms for the basic arithmetic operations over the integers immediately imply efficient algorithms for the corresponding arithmetic operations modulo N. For example, computing the modular reduction $[a \bmod N]$ can be done in time polynomial in $\|a\|$ and $\|N\|$ by simply computing division-with-remainder over the integers. Next, say $\|N\| = n$ and consider modular operations on two elements $a, b \in \mathbb{Z}_N$. (Note that a, b have length at most n. Actually, it is convenient to simply require that all elements of $\mathbb{Z}_N$ have length *exactly* n, padding to the left with 0s if necessary.) Addition of a and b modulo N can be done by first computing $a + b$, which is an integer of length at most $n + 1$, and then reducing this intermediate result modulo N. Similarly, multiplication modulo N can be performed by first computing the integer ab of length at most $2n$, and then reducing the result modulo N. Since addition, multiplication, and division-with-remainder can all be done in polynomial time, these give polynomial-time algorithms for addition and multiplication modulo N.

B.2.2 Computing Modular Inverses

Our discussion thus far has shown how to add, subtract, and multiply modulo N. One operation we are missing is "division" or, equivalently, computing multiplicative inverses modulo N. Recall from Section 7.1.2 that the multiplicative inverse (modulo N) of an element $a \in \mathbb{Z}_N$ is an element $a^{-1} \in \mathbb{Z}_N$ such that $a \cdot a^{-1} = 1 \bmod N$. Proposition 7.7 shows that a has an inverse if and only if $\gcd(a, N) = 1$, i.e., if and only if $a \in \mathbb{Z}_N^*$. Thus, using the Euclidean algorithm we can easily determine whether a given element a has a multiplicative inverse modulo N.

Given N and $a \in \mathbb{Z}_N$ with $\gcd(a, N) = 1$, Proposition 7.2 tells us that there exist integers X, Y with $Xa + YN = 1$. Recall that $[X \bmod N]$ is the multiplicative inverse of a; this holds since

$$[X \bmod N] \cdot a = Xa = 1 - YN = 1 \bmod N.$$

Integers X and Y satisfying $Xa + YN = 1$ can be found efficiently using the extended Euclidean algorithm eGCD shown in Section B.1.2. This leads to the following polynomial-time algorithm for computing multiplicative inverses:

ALGORITHM B.11
Computing modular inverses

Input: Modulus N; element a
Output: $[a^{-1} \bmod N]$ (if it exists)

$(d, X, Y) := \mathsf{eGCD}(a, N)$ /* note that $Xa + YN = \gcd(a, N)$ */
if $d \neq 1$ **return** "a is not invertible modulo N"
else return $[X \bmod N]$

B.2.3 Modular Exponentiation

A more challenging task is that of *exponentiation* modulo N; that is, computing $[a^b \bmod N]$ for base $a \in \mathbb{Z}_N$ and integer exponent $b > 0$. (When $b = 0$ the problem is easy. When $b < 0$ and $a \in \mathbb{Z}_N^*$ then $a^b = (a^{-1})^{-b} \bmod N$ and the problem is reduced to the case of exponentiation with a positive exponent given that we can compute inverses as discussed in the previous section.) Notice that the basic approach used in the case of addition and multiplication (i.e., computing the integer a^b and then reducing this intermediate result modulo N) *does not work* here: the integer a^b has length $\|a^b\| = \Theta(\log a^b) = \Theta(b \cdot \|a\|)$, and so even storing the intermediate result a^b would require time that is exponential in $\|b\| = \Theta(\log b)$.

We can address this problem by reducing modulo N repeatedly throughout the process of computing the result, rather than waiting until the end to reduce modulo N. This has the effect of keeping the intermediate results "small" throughout the computation. Even with this important initial obser-

vation, it is still non-trivial to design a polynomial-time algorithm for modular exponentiation. Consider the following naïve approach:

ALGORITHM B.12
A naïve algorithm for modular exponentiation

Input: Modulus N; base $a \in \mathbb{Z}_N$; integer exponent $b > 0$
Output: $[a^b \bmod N]$

$x := 1$
for $i = 1$ to b:
  $x := x \cdot a \bmod N$
return x

Since this algorithm uses b iterations of the inner loop, it still runs in time that is *exponential* in $\|b\|$.

The naïve algorithm given above can be viewed as relying on the following recurrence:

$$[a^b \bmod N] = [a \cdot a^{b-1} \bmod N] = [a \cdot a \cdot a^{b-2} \bmod N] = \cdots,$$

and could easily have been written recursively in which case the correspondence would be even more clear. Looking at the above equation, we see that any algorithm depending on this recurrence will require $\Theta(b)$ time. We can do better by making use of the following recurrence:

$$[a^b \bmod N] = \begin{cases} \left[\left(a^{\frac{b}{2}}\right)^2 \bmod N\right] & \text{when } b \text{ is even} \\ \left[a \cdot \left(a^{\frac{b-1}{2}}\right)^2 \bmod N\right] & \text{when } b \text{ is odd.} \end{cases}$$

Following this approach leads to a recursive algorithm — called, for obvious reasons, "square-and-multiply" (or sometimes "repeated squaring") — that requires only $\mathcal{O}(\log b) = \mathcal{O}(\|b\|)$ modular squarings/multiplications; see Algorithm B.13. In the algorithm, the length of b decreases by 1 in each recursive call; it follows that the number of recursive calls is at most $\|b\|$. Furthermore, the operations carried out during each recursive call can be performed in time polynomial in $\|a\|$ and $\|N\|$. It follows that the algorithm as a whole runs in time polynomial in $\|a\|$, $\|b\|$, and $\|N\|$. Looking carefully at the algorithm, we see that it performs at most $2 \cdot \|b\|$ multiplications-plus-reductions modulo N. Algorithm B.13 is written recursively for ease of understanding. In practice, for reasons of efficiency, an iterative algorithm is preferred. See Exercise B.3.

Fix a and N and consider the modular exponentiation function given by $f_{a,N}(b) = [a^b \bmod N]$. We have just seen that computing $f_{a,N}$ is easy. In contrast, computing the *inverse* of this function — that is, computing b given a, N, and $[a^b \bmod N]$ — is believed to be hard for appropriate choice of a

ALGORITHM B.13
Algorithm ModExp **for efficient modular exponentiation**

Input: Modulus N; base $a \in \mathbb{Z}_N$; integer exponent $b > 0$
Output: $[a^b \bmod N]$

if $b = 1$ **return** a
else
  if b is even
    $t := \mathsf{ModExp}\,(N, a, b/2)$
    return $[t^2 \bmod N]$
  if b is odd
    $t := \mathsf{ModExp}\,(N, a, (b-1)/2)$
    return $[a \cdot t^2 \bmod N]$

and N. Inverting the modular exponentiation function is known as solving the *discrete logarithm problem*, something we discuss in detail in Section 7.3.2.

Exponentiation in Arbitrary Groups

The efficient modular exponentiation algorithm given above carries over in a straightforward way to enable efficient exponentiation in any group, as long as the underlying group operation can be performed efficiently. Specifically, if $\mathbb{G}$ is a group and g is an element of $\mathbb{G}$, then g^b can be computed using at most $2 \cdot \|b\|$ applications of the underlying group operation.

If the order q of $\mathbb{G}$ is known, then $a^b = a^{[b \bmod q]}$ (cf. Proposition 7.49) and this can be used to further speed up the computation by reducing b modulo q first. This remark applies also to the modular exponentiation algorithms described earlier.

Considering the (additive) group $\mathbb{Z}_N$, the group exponentiation algorithm just described gives a method for computing the "exponentiation"

$$[b \cdot g \bmod N] \stackrel{\text{def}}{=} [\underbrace{g + \cdots + g}_{b \text{ times}} \bmod N]$$

that differs from the method discussed earlier that relies on standard integer multiplication followed by a modular reduction. In comparing the two approaches to solving the same problem, note that the original algorithm uses specific information about $\mathbb{Z}_N$; in particular, it (essentially) treats the "exponent" b as an element of $\mathbb{Z}_N$ (possibly by reducing b modulo N first). In contrast, the "square-and-multiply" algorithm just presented treats $\mathbb{Z}_N$ only as an abstract group. (Of course, the group operation of addition modulo N relies on the specifics of $\mathbb{Z}_N$.)

The point of this discussion is merely to illustrate that some group algorithms are *generic* (i.e., they apply equally well to all groups) while some group algorithms rely on specific properties of a *particular* group or class of groups. We saw some examples of this phenomenon in Chapter 8.

B.2.4 Choosing a Random Group Element

For cryptographic applications, it is often required to choose a random element of a given group $\mathbb{G}$. (Recall our convention that "random" means "uniformly distributed.") We first treat the problem in an abstract group, and then focus specifically on the cases of $\mathbb{Z}_N$ and $\mathbb{Z}_N^*$.

Elements of a group $\mathbb{G}$ must be specified using some *representation* of these elements as bit-strings, where we assume without any real loss of generality that the elements of a given group are all represented using strings of the same length. (It is also crucial, especially for our discussion in this section, that there is a *unique* string representing each group element.) For example, if $\|N\| = n$ then elements of $\mathbb{Z}_N$ can all be represented as strings of length n, where the integer $a \in \mathbb{Z}_N$ is simply padded to the left with 0s in case $\|a\| < n$.

We do not focus much on the issue of representation, since for all the groups considered in this text the representation can simply be taken to be the "natural" one (as in the case of $\mathbb{Z}_N$, above). Note, however, that different representations of the same group can affect the complexity of performing various computations, and so choosing the "right" representation for a given group is often important in practice. Since our goal is only to show *polynomial-time* algorithms for each of the operations we need (and not to show the most efficient algorithms known), the exact representation used is less important for our purposes. Moreover, most of the "higher-level" algorithms we present use the group operation in a "black-box" manner, so that as long as the group operation can be performed in polynomial time (in some parameter), the resulting algorithm will run in polynomial time as well.

Given a group $\mathbb{G}$ where elements are represented by strings of length ℓ, a random group element can be selected by choosing random ℓ-bit strings until a group element is found. To obtain an algorithm with bounded running time, we introduce a parameter t bounding the maximum number of times this process is repeated; if all t iterations fail to select an element of $\mathbb{G}$, then the algorithm outputs fail. (An alternative is to output an arbitrary element of $\mathbb{G}$.) That is:

ALGORITHM B.14
Choosing a random group element

Input: A (description of a) group $\mathbb{G}$; length-parameter ℓ; parameter t
Output: A random element of $\mathbb{G}$

for $i = 1$ to t:
  $x \leftarrow \{0,1\}^\ell$
  if $x \in \mathbb{G}$ **return** x
return "fail"

It is fairly obvious that whenever the above algorithm does *not* output fail, it outputs a uniformly-distributed element of $\mathbb{G}$. This is simply because each element of $\mathbb{G}$ is equally likely to be chosen in any given iteration. Formally, if we let Fail denote the event that the algorithm outputs fail, then for any element $g \in \mathbb{G}$ we have

$$\Pr\left[\text{output of the algorithm equals } g \mid \overline{\mathsf{Fail}}\right] = \frac{1}{|\mathbb{G}|}.$$

What is the probability that the algorithm outputs fail? In any given iteration the probability that $x \in \mathbb{G}$ is exactly $|\mathbb{G}|/2^\ell$, and so the probability that x does *not* lie in $\mathbb{G}$ in any of the t iterations is

$$\left(1 - \frac{|\mathbb{G}|}{2^\ell}\right)^t. \tag{B.1}$$

In cryptographic settings, there will be a security parameter n and the group $\mathbb{G}$ (as well as ℓ) will depend on n rather than being fixed. Formally, we fix some *class* $\mathcal{C}$ of groups (rather than a single group), associate a value n with each group in the class, and ask whether it is possible to sample a random element from a group $\mathbb{G} \in \mathcal{C}$ in time polynomial in the parameter n associated with $\mathbb{G}$. That is, we ask whether it is possible to sample a random element in polynomial time from the groups in the class $\mathcal{C}$.

(As a technical note, the class also specifies a representation for each group $\mathbb{G}$ in the class. We require $\ell = \mathsf{poly}(n)$ and assume that all groups sharing a particular value of n have the same length-parameter $\ell = \ell(n)$.)

As an example, we can consider the class of groups $\mathcal{C} = \{\mathbb{Z}_N \mid N \in \mathbb{Z}\}$, with parameter $n = \|N\|$ associated with the group $\mathbb{Z}_N$ in this class. Then the question is whether it is possible to sample a random element from $\mathbb{Z}_N$ in $\mathsf{poly}(\|N\|)$ time.

There is a trade-off between the running time of Algorithm B.14 and the probability that the algorithm outputs fail, since increasing t decreases the probability of failure but increases the worst-case running time. For cryptographic applications we need an algorithm where the worst-case running time is polynomial in n, while the failure probability is negligible in n. To achieve this, two conditions must hold for each group $\mathbb{G}$ (with parameter n) in the class:

1. It should be possible to determine in $\mathsf{poly}(n)$ time whether an $\ell(n)$-bit string is an element of $\mathbb{G}$ or not; and
2. the probability that a random $\ell(n)$-bit string is an element of $\mathbb{G}$ should be at least $1/\mathsf{poly}(n)$.

The need for the first condition is obvious, since Algorithm B.14 needs to check whether $x \in \mathbb{G}$. Say the second condition holds, i.e., there is a polynomial p such that for every group $\mathbb{G}$ (in the given class $\mathcal{C}$) with associated parameter n

and length-parameter $\ell = \ell(n)$, the probability that a random $\ell(n)$-bit string is an element of $\mathbb{G}$ is at least $1/p(n)$. Set $t(n) = \lceil p(n) \cdot n \rceil$. Then the probability that the algorithm outputs fail is (cf. Equation (B.1)):

$$\begin{aligned}\left(1 - \frac{|\mathbb{G}|}{2^\ell}\right)^t &\leq \left(1 - \frac{1}{p(n)}\right)^{\lceil p(n) \cdot n \rceil} \\ &\leq \left(\left(1 - \frac{1}{p(n)}\right)^{p(n)}\right)^n \\ &\leq \left(e^{-1}\right)^n \;=\; e^{-n}\,,\end{aligned}$$

using Proposition A.2 for the third inequality. We thus see that when the second condition holds, it is possible to obtain an algorithm with $t = \mathsf{poly}(n)$ and failure probability negligible in n.

We stress that the two conditions given above are not guaranteed to hold for arbitrary classes of groups. Instead, they must be verified for each particular class of interest.

The Case of $\mathbb{Z}_N$

Consider groups of the form $\mathbb{Z}_N$, with $n = \|N\|$. It is easy to verify each of the conditions outlined previously. Checking whether an n-bit string x (interpreted as a positive integer of length at most n) is an element of $\mathbb{Z}_N$ simply requires checking whether $x < N$, which can clearly be done in $\mathsf{poly}(n)$ time. Furthermore, the probability that a random n-bit string lies in $\mathbb{Z}_N$ is

$$\frac{N}{2^n} \geq \frac{2^{n-1}}{2^n} = \frac{1}{2}\,.$$

For concreteness, we show the algorithm resulting from the above:

ALGORITHM B.15
Choosing a random element of $\mathbb{Z}_N$

Input: Modulus N of length n
Output: A random element of $\mathbb{Z}_N$

for $i = 1$ to $2n$:
  $x \leftarrow \{0,1\}^n$
  if $x < N$ **return** x
return "fail"

This algorithm runs in $\mathsf{poly}(n)$ time, and outputs fail with probability negligible in n.

The Case of $\mathbb{Z}_N^*$

Consider next groups of the form $\mathbb{Z}_N^*$, with $n = \|N\|$ as before. We leave it to the exercises to show how to determine whether an n-bit string x is an element of $\mathbb{Z}_N^*$ or not. To prove the second condition, we need to show that $\frac{\phi(N)}{2^n} \geq 1/\mathsf{poly}(n)$. Since

$$\frac{\phi(N)}{2^n} = \frac{N}{2^n} \cdot \frac{\phi(N)}{N},$$

and we have already seen that $\frac{N}{2^n} \geq \frac{1}{2}$, the desired bound is a consequence of the following theorem.

THEOREM B.16 *For $N \geq 3$ of length n, we have $\frac{\phi(N)}{N} > 1/2n$.*

(Stronger bounds are known, but the above suffices for our purpose.) We do not prove the theorem, but instead content ourselves with lower-bounding $\phi(N)/N$ in two special cases: when N is prime and when N is a product of two close-to-equal-length (distinct) primes.

The analysis is easy when N is an odd prime. Here $\phi(N) = N - 1$ and, as when we analyzed the algorithm for choosing a random element of $\mathbb{Z}_N$,

$$\frac{\phi(N)}{2^n} = \frac{N-1}{2^n} \geq \frac{2^{n-1}}{2^n} = \frac{1}{2}.$$

Consider next the case of $N = pq$ for p and q distinct primes each of length roughly $n/2$.

PROPOSITION B.17 *Let $N = pq$ where p and q are distinct primes each of length at least $n/2$. Then $\frac{\phi(N)}{N} = 1 - \mathsf{negl}(n)$.*

PROOF We have

$$\frac{\phi(N)}{N} = \frac{(p-1)(q-1)}{pq} = 1 - \frac{1}{q} - \frac{1}{p} + \frac{1}{pq} > 1 - \frac{1}{q} - \frac{1}{p} \geq 1 - 2 \cdot 2^{-(\frac{n}{2}-1)}.$$

The proposition follows. ■

We conclude that when N is prime or the product of two distinct large primes, there exists an algorithm for generating a random element of $\mathbb{Z}_N^*$ that runs in time polynomial in $n = \|N\|$ and outputs fail with probability negligible in n.

In this book, we simply write "$x \leftarrow \mathbb{Z}_N$" or "$x \leftarrow \mathbb{Z}_N^*$" to denote random selection of an element x from $\mathbb{Z}_N$ or $\mathbb{Z}_N^*$ using, e.g., one of the algorithms of this section. We stress that we will simply assume that x lies in the desired range, with the implicit understanding that the algorithm for choosing x may output fail with negligible probability.

B.3 * Finding a Generator of a Cyclic Group

In this section we will be concerned with the problem of finding a generator of an arbitrary cyclic group $\mathbb{G}$ of order q. Here, q does not necessarily denote a prime number; indeed, the problem of finding a generator when q is prime is rendered trivial by Corollary 7.52.

Our approach to finding a generator will be to find a *random* generator, proceeding in a manner very similar to that of Section B.2.4. Namely, we will repeatedly sample random elements of $\mathbb{G}$ until we find an element that is a generator. As in Section B.2.4, an analysis of this method requires an understanding of two things:

- How to efficiently test whether a given element is a generator; and
- the fraction of group elements that are generators.

In order to understand these issues, we first develop a bit of additional group-theoretic background.

B.3.1 Group-Theoretic Background

Recall that the order of an element h is the smallest positive integer i for which $h^i = 1$. Let g be a generator of a group $\mathbb{G}$ of order $q > 1$; note that this means the order of g is q. Consider any element $h \in \mathbb{G}$ that is not the identity (the identity cannot be a generator of $\mathbb{G}$), and let us ask whether this element might also be a generator of $\mathbb{G}$. Since g generates $\mathbb{G}$, we can write $h = g^x$ for some $x \in \{1, \ldots, q-1\}$ (note $x \neq 0$ since h is not the identity). Consider two cases:

Case 1: $\gcd(x, q) = r > 1$. Write $x = \alpha \cdot r$ and $q = \beta \cdot r$ with α, β non-zero integers less than q. Then:

$$h^\beta = (g^x)^\beta = g^{\alpha r \beta} = (g^q)^\alpha = 1.$$

So the order of h is at most $\beta < q$, and h cannot be a generator of $\mathbb{G}$.

Case 2: $\gcd(x, q) = 1$. Let $i \leq q$ be the order of h. Then

$$g^0 = 1 = h^i = (g^x)^i = g^{xi},$$

implying $xi = 0 \bmod q$ by Proposition 7.50. This means that $q \,|\, xi$. Since $\gcd(x, q) = 1$, however, Proposition 7.3 shows that $q \,|\, i$ and so $i = q$. We conclude that h is a generator of $\mathbb{G}$.

Summarizing the above, we see that for $x \in \{1, \ldots, q-1\}$ the element $h = g^x$ is a generator of $\mathbb{G}$ exactly when $\gcd(x, q) = 1$. We have seen the set

$\{x \in \{1, \ldots, q-1\} \mid \gcd(x, q) = 1\}$ before — it is exactly $\mathbb{Z}_q^*$! We have thus proved the following:

THEOREM B.18 *Let $\mathbb{G}$ be a cyclic group of order $q > 1$ with generator g. Then there are $\phi(q)$ generators of $\mathbb{G}$, and these are exactly given by the set $\{g^x \mid x \in \mathbb{Z}_q^*\}$.*

In particular, if $\mathbb{G}$ is a group of prime order q, then it has $\phi(q) = q - 1$ generators exactly in agreement with Corollary 7.52.

We turn next to the question of determining whether a given element h is a generator of $\mathbb{G}$. Of course, one way to check whether h generates $\mathbb{G}$ is to simply enumerate $\{h^0, h^1, \ldots, h^{q-1}\}$ to see whether this list includes every element of $\mathbb{G}$. This requires time linear in q (i.e., exponential in $\|q\|$) and is therefore unacceptable for our purposes. Another approach, if we already know a generator g, is to compute the discrete logarithm $x = \log_g h$ and then apply the previous theorem; in general, however, we may not have such a g, and anyway computing the discrete logarithm may itself be a hard problem.

If we know the factorization of q, we can do better.

PROPOSITION B.19 *Let $\mathbb{G}$ be a group of order q, and let $q = \prod_{i=1}^{k} p_i^{e_i}$ be the prime factorization of q, where the $\{p_i\}$ are distinct primes and $e_i \geq 1$. Set $q_i = q/p_i$. Then $h \in \mathbb{G}$ is a generator of $\mathbb{G}$ if and only if*

$$h^{q_i} \neq 1 \quad \textit{for } i = 1, \ldots, k.$$

PROOF One direction is easy. Say $h^{q_i} = 1$ for some i. Then the order of h is at most $q_i < q$, and so h cannot be a generator.

Conversely, say h is not a generator but instead has order $q' < q$. By Proposition 7.51, we know $q' \mid q$. This implies that q' can be written as $q' = \prod_{i=1}^{k} p_i^{e_i'}$, where $e_i' \geq 0$ and for at least one index j we have $e_j' < e_j$. But then q' divides $q_j = p_j^{e_j - 1} \cdot \prod_{i \neq j} p_i^{e_i}$, and so (using Proposition 7.50) $h^{q_j} = h^{[q_j \bmod q']} = h^0 = 1$. ∎

The proposition does not require $\mathbb{G}$ to be cyclic; if $\mathbb{G}$ is not cyclic then every element $h \in \mathbb{G}$ will satisfy $h^{q_i} = 1$ for some i and so there are no generators (as must be the case if $\mathbb{G}$ is not cyclic).

B.3.2 Efficient Algorithms

We now show how to efficiently *test* whether a given element is a generator, as well as how to efficiently *find* a generator in an arbitrary group.

Testing if an Element is a Generator

Proposition B.19 immediately suggests an efficient algorithm for deciding whether a given element h is a generator or not.

ALGORITHM B.20
Testing whether an element is a generator

Input: Group order q; prime factors $\{p_i\}_{i=1}^{k}$ of q; element $h \in \mathbb{G}$
Output: A decision as to whether h is a generator of $\mathbb{G}$

for $i = 1$ to k:
  if $h^{q/p_i} = 1$ **return** "h is not a generator"
return "h is a generator"

Correctness of the algorithm is evident from Proposition B.19. We now show that the algorithm terminates in time polynomial in $\|q\|$. Since, in each iteration, h^{q/p_i} can be computed in polynomial time, we need only show that the number of iterations k is polynomial. This is the case since an integer q can have no more than $\log_2 q = \mathcal{O}(\|q\|)$ prime factors; this is true because

$$q = \prod_{i=1}^{k} p_i^{e_i} \geq \prod_{i=1}^{k} p_i \geq \prod_{i=1}^{k} 2 = 2^k$$

and so $k \leq \log_2 q$.

Algorithm B.20 requires the prime factors of the group order q to be provided as input. Interestingly, there is no known efficient algorithm for testing whether an element of an arbitrary group is a generator when the factors of the group order are *not* known.

The Fraction of Elements that are Generators

As shown in Theorem B.18, the fraction of elements of a group $\mathbb{G}$ of order q that are generators is $\phi(q)/q$. Theorem B.16 says that $\phi(q)/q = \Omega(1/\|q\|)$. The fraction of elements that are generators is thus sufficiently high to ensure that sampling a polynomial number of elements from the group will yield a generator with all but negligible probability. (The analysis is the same as in Section B.2.4.)

Concrete Examples in $\mathbb{Z}_p^*$

Putting everything together, there is an efficient probabilistic method for finding a generator of a group $\mathbb{G}$ *as long as the factorization of the group order is known.* When selecting a group for cryptographic applications, it is

therefore important that the group is chosen in such a way that this holds. For groups of the form $\mathbb{Z}_p^*$, with p prime, some of the possibilities are as follows:

- As we have already discussed fairly extensively in Section 7.3.2, working in a prime order subgroup of $\mathbb{Z}_p^*$ has the effect of, among other things, eliminating the above difficulties that arise when q is not prime. Recall that one way to obtain such a subgroup is to choose p as a *strong* prime (i.e., so that $p = 2q+1$ with q also prime) and then work in the subgroup of quadratic residues modulo p (which is a subgroup of prime order q). In this case, all elements (apart from the identity) are generators.
- Alternately, if p is a strong prime as above then the order of the cyclic group $\mathbb{Z}_p^*$ is $2q$ and so the factorization of the group order is known. A generator of this group can thus be easily found, even though the group does not have prime order.
- Another possibility is to generate a random prime p in such a way that the factorization of $p-1$ is known. This is possible, but the details are beyond the scope of this book.

References and Additional Reading

The book by Shoup [131] is highly recommended for those seeking to explore the topics of this chapter in further detail. In particular, bounds on $\phi(N)/N$ (and an asymptotic version of Theorem B.16) can be found in [131, Chapter 5].

A nice result by Kalai [82] gives an easy method for generating random numbers along with their prime factorization.

Exercises

B.1 Prove correctness of the extended Euclidean algorithm.

B.2 Prove that the extended Euclidean algorithm runs in time polynomial in the lengths of its inputs.

> **Hint:** First prove a proposition analogous to Proposition B.8.

B.3 Develop an iterative algorithm for efficient (i.e., polynomial-time) computation of $[a^b \bmod N]$. (An iterative algorithm does not make recursive calls to itself.)

> **Hint:** Use auxiliary variables x (initialized to a) and t (initialized to 1), and maintain the invariant $t \cdot x^b = a^b \bmod N$. The algorithm terminates when $x = 1$ and t holds the final result.

B.4 Show how to determine that an n-bit string is in $\mathbb{Z}_N^*$ in polynomial time.

References

[1] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison Wesley, 2nd edition, 2002.

[2] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[3] W. Aiello, M. Bellare, G. Di Crescenzo, and R. Venkatesan. Security amplification by composition: The case of doubly-iterated, ideal ciphers. In *Advances in Cryptology — Crypto '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 390–407. Springer, 1998.

[4] A. Akavia, S. Goldwasser, and S. Safra. Proving hard-core predicates using list decoding. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, pages 146–157. IEEE, 2003.

[5] W. Alexi, B. Chor, O. Goldreich, and C.P. Schnorr. RSA and Rabin functions: Certain parts are as hard as the whole. *SIAM Journal on Computing*, 17(2):194–209, 1988.

[6] J. H. An, Y. Dodis, and T. Rabin. On the security of joint signature and encryption. In *Advances in Cryptology — Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 83–107. Springer, 2002.

[7] P. Barreto. The hashing function lounge. Online link can be found at `http://paginas.terra.com.br/informatica/paulobarreto`.

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology — Crypto '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[9] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE, 1997.

[10] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology — Crypto '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45. Springer, 1998.

[11] M. Bellare, O. Goldreich, and A. Mityagin. The power of verification queries in message authentication and authenticated encryption. Available at `http://eprint.iacr.org/2004/309`.

[12] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000.

[13] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology — Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[14] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *1st ACM Conf. on Computer and Communications Security*, pages 62–73. ACM, 1993.

[15] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology — Eurocrypt '94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 1995.

[16] M. Bellare and P. Rogaway. The exact security of digital signatures — how to sign with RSA and Rabin. In *Advances in Cryptology — Eurocrypt '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer, 1996.

[17] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology — Eurocrypt 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006. A full version of the paper is available at `http://eprint.iacr.org`.

[18] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[19] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[20] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Advances in Cryptology — Crypto '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[21] M. Blum. Coin flipping by telephone. In *Proc. IEEE COMPCOM*, pages 133–137, 1982.

[22] M. Blum and S. Goldwasser. An efficient probabilistic public-key encryption scheme which hides all partial information. In *Advances in Cryptology — Crypto '84*, volume 196 of *Lecture Notes in Computer Science*, pages 289–302. Springer, 1985.

[23] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[24] D. Boneh. The decision Diffie-Hellman problem. In *Algorithmic Number Theory, 3rd Intl. Symposium*, volume 1423 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 1998.

[25] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999.

[26] D. Boneh. Simplified OAEP for the RSA and Rabin functions. In *Advances in Cryptology — Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 275–291. Springer, 2001.

[27] D. Boneh, A. Joux, and P. Nguyen. Why textbook ElGamal and RSA encryption are insecure. In *Advances in Cryptology — Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2000.

[28] D. Boneh and R. Venkatesan. Breaking RSA may not be equivalent to factoring. In *Advances in Cryptology — Eurocrypt '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 59–71. Springer, 1998.

[29] D. Bressoud. *Factorization and Primality Testing*. Springer, 1989.

[30] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[31] D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In *Advances in Cryptology — Crypto '91*, volume 576 of *Lecture Notes in Computer Science*, pages 470–484. Springer, 1992.

[32] L.N. Childs. *A Concrete Introduction to Higher Algebra*. Undergraduate Texts in Mathematics. Springer, 2nd edition, 2000.

[33] C. Cid, S. Murphy, and M. Robshaw. *Algebraic Aspects of the Advanced Encryption Standard*. Springer, 2006.

[34] D. Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, 1994. Available for download from `http://researchweb.watson.ibm.com/journal/rd/`.

[35] J.-S. Coron. On the exact security of full-domain hash. In *Advances in Cryptology — Crypto 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 229–235. Springer, 2000.

[36] J.-S. Coron. Optimal security proofs for PSS and other signature schemes. In *Advances in Cryptology — Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2002.

[37] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In *Advances in Cryptology*

— *Crypto 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[38] R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. *ACM Transactions on Information and System Security*, 3(3):161–185, 2000.

[39] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[40] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer, 2nd edition, 2005.

[41] J. Daemen and V. Rijmen. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer, 2002.

[42] W. Dai. Crypto++ 5.2.1 benchmarks. Available for download from `http://www.cryptopp.com/benchmarks.html`.

[43] I. Damgård. Collision free hash functions and public key signature schemes. In *Advances in Cryptology — Eurocrypt '87*, volume 304 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1988.

[44] I. Damgård. A design principle for hash functions. In *Advances in Cryptology — Crypto '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[45] J. DeLaurentis. A further weakness in the common modulus protocol for the RSA cryptoalgorithm. *Cryptologia*, 8:253–259, 1984.

[46] M. Dietzfelbinger. *Primality Testing in Polynomial Time*. Springer, 2004.

[47] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[48] W. Diffie and M. Hellman. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, pages 74–84, June 1977.

[49] J.D. Dixon. Asymptotically fast factorization of integers. *Mathematics of Computation*, 36:255–260, 1981.

[50] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

[51] C. Ellison and B. Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[52] S. Even and Y. Mansour. A construction of a cipher from a single pseudorandom permutation. In *ASIACRYPT*, volume 739 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 1993.

[53] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, 1973.

[54] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — Crypto '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1987.

[55] M. Fischlin. The Cramer-Shoup strong-RSA signature scheme revisited. In *Public Key Cryptography — PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 2003.

[56] S. Fluhrer, I. Mantin, and A. Shamir. Attacks on RC4 and WEP. *CryptoBytes*, 5(2):26–34, 2002.

[57] J.B. Fraleigh. *A First Course in Abstract Algebra*. Addison Wesley, 7th edition, 2002.

[58] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology — Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2001.

[59] T. El Gamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Info. Theory*, 31(4):469–472, 1985.

[60] C.F. Gauss. *Disquisitiones Arithmeticae*. Springer, 1986. (English edition).

[61] R. Gennaro, S. Halevi, and T. Rabin. Secure hash-and-sign signatures without the random oracle. In *Advances in Cryptology — Eurocrypt '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 1999.

[62] E.-J. Goh, S. Jarecki, J. Katz, and N. Wang. Efficient signature schemes with tight security reductions to the Diffie-Hellman problems. *J. Cryptology*, to appear.

[63] O. Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In *Advances in Cryptology — Crypto '86*, volume 263 of *Lecture Notes in Computer Science*, pages 104–110. Springer, 1987.

[64] O. Goldreich. *Foundations of Cryptography, vol. 1: Basic Tools*. Cambridge University Press, 2001.

[65] O. Goldreich. *Foundations of Cryptography, vol. 2: Basic Applications*. Cambridge University Press, 2004.

[66] O. Goldreich, S. Goldwasser, and S. Micali. On the cryptographic applications of random functions. In *Advances in Cryptology – Crypto'84*,

volume 196 of *Lecture Notes in Computer Science*, pages 276–288. Springer, 1985.

[67] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986.

[68] O. Goldreich and L. Levin. A hard-core predicate for all one-way functions. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 25–32. ACM, 1989.

[69] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[70] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, 1988.

[71] S. Goldwasser, S. Micali, and A.C.-C. Yao. Strong signature schemes. In *Proc. 15th Annual ACM Symposium on Theory of Computing*, pages 431–439. ACM, 1983.

[72] D. Hankerson, A.J. Menezes, and S.A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[73] J. Håstad. Solving simultaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2):336–341, 1988.

[74] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[75] J. Håstad and M. Näslund. The security of all RSA and discrete log bits. *Journal of the ACM*, 51(2):187–230, 2004.

[76] I.N. Herstein. *Abstract Algebra*. Wiley, 3rd edition, 1996.

[77] H. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3):189–221, 2002.

[78] H.M. Heys. *The Design of Substitution-Permutation Network Ciphers Resistant to Cryptanalysis*. PhD thesis, Queen's University, 1994.

[79] R. Impagliazzo and M. Luby. One-way functions are essential for complexity based cryptography. In *Proc. 30th Annual Symposium on Foundations of Computer Science*, pages 230–235. IEEE, 1989.

[80] ISO/IEC 9797. Data cryptographic techniques — data integrity mechanism using a cryptographic check function employing a block cipher algorithm, 1989.

[81] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996.

[82] A. Kalai. Generating random factored numbers, easily. *Journal of Cryptology*, 16(4):287–289, 2003.

[83] J. Katz. *Digital Signatures*. Springer, 2007.

[84] J. Katz and C.-Y. Koo. On constructing universal one-way hash functions from arbitrary one-way functions. *J. Cryptology*, to appear. Available at `http://eprint.iacr.org/2005/328`.

[85] J. Katz and M. Yung. Unforgeable encryption and chosen-ciphertext secure modes of operation. In *Fast Software Encryption — FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2001.

[86] J. Katz and M. Yung. Characterization of security notions for probabilistic private-key encryption. *Journal of Cryptology*, 19(1):67–96, 2006.

[87] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 2nd edition, 2002.

[88] J. Kilian and P. Rogaway. How to protect DES against exhaustive key search (an analysis of DESX). *Journal of Cryptology*, 14(1):17–35, 2001.

[89] L. Kohnfelder. Towards a practical public-key cryptosystem, 1978. Undergraduate thesis, MIT.

[90] H. Krawczyk. The order of encryption and authentication for protecting communication (or: How secure is SSL?). In *Advances in Cryptology — Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer, 2001.

[91] H. Kugel. America's code breaker. Available for download from: `http://militaryhistory.about.com/`.

[92] L. Lamport. Constructing digital signatures from a one-way function. Technical Report CSL-98, SRI International, 1978.

[93] S.K. Langford. *Differential-Linear Cryptanalysis and Threshold Signatures*. PhD thesis, Stanford University, 1995.

[94] A.K. Lenstra and E.R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[95] S. Levy. *Crypto: How the Code Rebels Beat the Government — Saving Privacy in the Digital Age*. Viking, 2001.

[96] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996.

[97] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[98] M. Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology — Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*. Springer, 1994.

[99] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRS Press, 1997.

[100] R. Merkle. A digital signature scheme based on a conventional encryption function. In *Advances in Cryptology — Crypto '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1988.

[101] R. Merkle. A certified digital signature. In *Advances in Cryptology — Crypto '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1990.

[102] R. Merkle. One way hash functions and DES. In *Advances in Cryptology — Crypto '89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[103] R. Merkle and M. Hellman. On the security of multiple encryption. *Communications of the ACM*, 24(7):465–467, 1981.

[104] S. Micali, C. Rackoff, and B. Sloan. The notion of security for probabilistic cryptosystems. *SIAM J. Computing*, 17(2):412–426, 1988.

[105] G.L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[106] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 33–43. ACM, 1989.

[107] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 427–437. ACM, 1990.

[108] National Bureau of Standards. DES modes of operation, 1980. Federal Information Processing Standard (FIPS), publication 81.

[109] National Bureau of Standards. Data encryption standard (DES), 1977. Federal Information Processing Standard (FIPS), publication 46.

[110] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC), 2002. Federal Information Processing Standard (FIPS), publication 198.

[111] V.I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

[112] A.M. Odlyzko. Discrete logarithms: The past and the future. *Designs, Codes, and Cryptography*, 19(2/3):129–145, 2000.

[113] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology — Eurocrypt '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[114] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. Information Theory*, 24(1):106–110, 1978.

[115] J.M. Pollard. Theorems of factorization and primality testing. *Proc. Cambridge Philosophical Society*, 76:521–528, 1974.

[116] J.M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[117] C. Pomerance. The quadratic sieve factoring algorithm. In *Advances in Cryptology — Eurocrypt '84*, volume 209 of *Lecture Notes in Computer Science*, pages 169–182. Springer, 1985.

[118] M.O. Rabin. Digitalized signatures. In R.A. Demillo, D.P. Dobkin, A.K. Jones, and R.J. Lipton, editors, *Foundations of Security Computation*, pages 155–168. Academic Press, 1978.

[119] M.O. Rabin. Digitalized signatures as intractable as factorization. Technical Report TR-212, MIT/LCS, 1979.

[120] M.O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[121] C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology — Crypto '91*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444. Springer, 1992.

[122] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[123] P. Rogaway and T. Shrimpton. Cryptographic hash function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption — FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[124] J. Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 387–394. ACM, 1990.

[125] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, 2nd edition, 1995.

[126] D. Shanks. Class number, a theory of factorization, and genera. In *Proc. Symposia in Pure Mathematics 20*, pages 415–440, 1971.

[127] C.E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[128] V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology — Eurocrypt '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer, 1997.

[129] V. Shoup. Why chosen ciphertext security matters. Technical Report RZ 3076, IBM Zurich, November 1998. Available at `http://shoup.net/papers/expo.pdf`.

[130] V. Shoup. OAEP reconsidered. In *Advances in Cryptology — Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 239–259. Springer, 2001.

[131] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2005. Also available at `http://www.shoup.net/ntb`.

[132] J.H. Silverman and J. Tate. *Rational Points on Elliptic Curves*. Undergraduate Texts in Mathematics. Springer, 1994.

[133] G. Simmons. A 'weak' privacy protocol using the RSA crypto algorithm. *Cryptologia*, 7:180–182, 1983.

[134] G. Simmons. A survey of information authentication. In G. Simmons, editor, *Contemporary Cryptology: The Science of Information Integrity*, pages 379–419. IEEE Press, 1992.

[135] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[136] D.R. Stinson. Universal hashing and authentication codes. *Designs, Codes, and Cryptography*, 4(4):369–380, 1994.

[137] D.R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, 1st edition, 1995.

[138] D.R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, 3rd edition, 2005.

[139] W. Trappe and L. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2nd edition, 2005.

[140] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[141] G.S. Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the American Institute for Electrical Engineers*, 55:109–115, 1926.

[142] S.S. Wagstaff, Jr. *Cryptanalysis of Number Theoretic Ciphers*. Chapman & Hall/CRC Press, 2003.

[143] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology — Crypto 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[144] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Advances in Cryptology — Eurocrypt 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[145] L. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/CRC Press, 2003.

[146] P. Weadon. The battle of Midway: AF is short of water, 2000. NSA Historical Publications. Available at: `http://www.nsa.gov` under Historical Publications.

[147] H. Wu. The misuse of RC4 in Microsoft Word and Excel. Available at `http://eprint.iacr.org/2005/007`.

[148] ANSI X9.9. American national standard for financial institution message authentication (wholesale), 1981.

[149] A.C.-C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd Annual Symposium on Foundations of Computer Science*, pages 80–91. IEEE, 1982.

Computer Science/Cryptography

INTRODUCTION TO MODERN CRYPTOGRAPHY

Cryptography plays a key role in ensuring the privacy and integrity of data and the security of computer networks. **Introduction to Modern Cryptography** provides a rigorous yet accessible treatment of modern cryptography, with a focus on formal definitions, precise assumptions, and rigorous proofs.

The authors introduce the core principles of modern cryptography, including the modern, computational approach to security that overcomes the limitations of perfect secrecy. An extensive treatment of private-key encryption and message authentication follows. The authors also illustrate design principles for block ciphers, such as the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES), and present provably secure constructions of block ciphers from lower-level primitives. The second half of the book focuses on public-key cryptography, beginning with a self-contained introduction to the number theory needed to understand the RSA, Diffie–Hellman, El Gamal, and other cryptosystems. After exploring public-key encryption and digital signatures, the book concludes with a discussion of the random oracle model and its applications.

Serving as a textbook, a reference, or for self-study, **Introduction to Modern Cryptography** presents the necessary tools to fully understand this fascinating subject.

Features

- Includes formal definitions, precise assumptions, and rigorous proofs
- Discusses many widely used cryptographic algorithms and standards
- Covers topics, such as pseudorandom generators/functions, Paillier encryption, and the random oracle model, often not found in other texts
- Contains suggestions for further reading as well as numerous exercises at the end of each chapter
- Assumes minimal prerequisites—all necessary mathematical background is included in the text

Chapman & Hall/CRC
Taylor & Francis Group
an informa business
www.taylorandfrancisgroup.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

C5513
ISBN 1-58488-551-3
90000
9 781584 885511

www.crcpress.com