Jonathan Katz and Yehuda Lindell

Introduction to Modern Cryptography

CRC PRESS
Boca Raton London New York Washington, D.C.

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 on the market — is that it provides a *rigorous* treatment of modern cryptography in an *accessible* manner appropriate for an introduction to the topic. To be sure, the material in this book is difficult (at least in comparison to some other books in this area). Rather than shy away from this difficulty, however, we have chosen to face it head-on, to lead the reader through the demanding (yet enthralling!) subject matter rather than shield the reader's eyes from it. We hope readers (and instructors) will respond by taking up the challenge.

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 Chapter 2, 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 assumptions*

must be clearly 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 *cryptographic constructions can be proven secure* with respect to a given definition of security and relative to a well-defined cryptographic assumption. This is the essence of modern cryptography, and was responsible for the transformation of 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 felt not only within the "theory of cryptography" community. The importance of precise definitions is, by now, widely understood and appreciated by those in the security community (as well as those 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 guide 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 useful.

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 7, introducing concrete mathematical problems believed to be "hard", providing the number-theoretic background needed to understand RSA, Diffie-Hellman, and El Gamal, and giving a flavor 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).
- 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 some 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. Alternately, 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 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 in the rare cases 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 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); Chapter 5 (discussing modern block ciphers and techniques used in their design); 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 material from Chapter 5 (see above) as well as Section 7.3.4 (elliptic-curve groups); Chapter 8 (algorithms for factoring and computing discrete logarithms); and Chapter 11 (describing the Goldwasser-Micali, Rabin, and Paillier encryption schemes along with all 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 is deeply indebted to Zvi Galil, Moti Yung, and Rafail Ostrovsky for their help, guidance, and support throughout his career. This book would never have come to be without their contributions to his development, and he thanks them for that. He would also like to thank his colleagues with whom he has had numerous discussions on the "right" approach to writing a cryptography textbook, and in particular Victor Shoup.

Yehuda Lindell wishes to first and foremost thank Oded Goldreich and Moni Naor for introducing him 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, he will just say *thank you* — you know who you are.

Both authors would like to extend their gratitude to those who read and commented on earlier drafts of this book. We thank Salil Vadhan and Alon Rosen who experimented with this text in an introductory course on cryptography at Harvard and provided us with valuable feedback. We also thank all of the following for their many comments and corrections: Adam Bender, Yair Dombb, William Glenn, S. Dov Gordon, Carmit Hazay, Avivit Levy, Matthew Mah, Jason Rogers, Rui Xue, Dicky Yan, and Hila Zarosim. We are very 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.

Contents

Preface **iii**

I Introduction and Classical Cryptography **1**

Rigorous

1 Introduction and Classical Ciphers **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 37
2.4 * Shannon's Theorem 38
2.5 Summary 40
References and Additional Reading 41
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 54
3.1.3 Proofs by Reduction 58
3.2 A Definition of Computationally-Secure Encryption 59
3.2.1 A Definition of Security for Encryption 60
3.2.2 * Properties of the Definition 64
3.3 Pseudorandomness 68
3.4 Constructing Secure Encryption Schemes 72
3.4.1 A Secure Fixed-Length Encryption Scheme 72

3.4.2 Handling Variable-Length Messages 75
3.4.3 Stream Ciphers and Multiple Encryptions 76
3.5 Security under Chosen-Plaintext Attacks (CPA) 81
3.6 Constructing CPA-Secure Encryption Schemes 85
3.6.1 Pseudorandom Functions 85
3.6.2 CPA-Secure Encryption Schemes from Pseudorandom Functions 88
3.6.3 Pseudorandom Permutations and Block Ciphers . . . 93
3.6.4 Modes of Operation 95
3.7 Security Against Chosen-Ciphertext Attacks (CCA) 100
References and Additional Reading 102
Exercises 103

4 Message Authentication Codes and Collision-Resistant Hash Functions 107
4.1 Secure Communication and Message Integrity 107
4.2 Encryption and Message Authentication 108
4.3 Message Authentication Codes – Definitions 109
4.4 Constructing Secure Message Authentication Codes 113
4.5 CBC-MAC 119
4.6 Collision-Resistant Hash Functions 121
4.6.1 Defining Collision Resistance 122
4.6.2 Weaker Notions of Security for Hash Functions 124
4.6.3 A Generic "Birthday" Attack 125
4.6.4 The Merkle-Damgård Transform 127
4.6.5 Collision-Resistant Hash Functions in Practice 129
4.7 * NMAC and HMAC 132
4.7.1 Nested MAC (NMAC) 132
4.7.2 HMAC 135
4.8 * Achieving Chosen-Ciphertext Secure Encryption 137
4.9 * Obtaining Privacy and Message Authentication 141
References and Additional Reading 147
Exercises 148

5 Pseudorandom Objects in Practice: Block Ciphers 151
5.1 Substitution-Permutation Networks 154
5.2 Feistel Networks 160
5.3 DES – The Data Encryption Standard 162
5.3.1 The Design of DES 162
5.3.2 Attacks on Reduced-Round Variants of DES 165
5.3.3 The Security of DES 168
5.4 Increasing the Key Size for Block Ciphers 170
5.5 AES – The Advanced Encryption Standard 173
5.6 Differential and Linear Cryptanalysis – A Brief Look 176
5.7 Stream Ciphers from Block Ciphers 177

Additional Reading and References 178
Exercises 179

6 * Theoretical Constructions of Pseudorandom Objects 181
6.1 One Way Functions 182
6.1.1 Definitions 182
6.1.2 Candidates 185
6.1.3 Hard-Core Predicates 186
6.2 Overview of Constructions 188
6.3 Hard-Core Predicates from Every One-Way Function 190
6.3.1 The Most Simplistic Case 190
6.3.2 A More Involved Case 191
6.3.3 The Full Proof 194
6.4 Constructions of Pseudorandom Generators 201
6.4.1 Pseudorandom Generators with Minimal Expansion . 201
6.4.2 Increasing the Expansion Factor 203
6.5 Constructions of Pseudorandom Functions 208
6.6 Constructions of Pseudorandom Permutations 212
6.7 Private-Key Cryptography – Necessary and Sufficient Assumptions 214
6.8 A Digression – Computational Indistinguishability 220
6.8.1 Pseudorandomness and Pseudorandom Generators . . 221
6.8.2 Multiple Samples 222
References and Additional Reading 225
Exercises 226

III Public-Key (Asymmetric) Cryptography 229

7 Number Theory and Cryptographic Hardness Assumptions 231
7.1 Preliminaries and Basic Group Theory 233
7.1.1 Primes and Divisibility 233
7.1.2 Modular Arithmetic 235
7.1.3 Groups 237
7.1.4 The Group $\mathbb{Z}_N^*$ and the Chinese Remainder Theorem . 241
7.1.5 Using the Chinese Remainder Theorem 245
7.2 Primes, Factoring, and RSA 248
7.2.1 Generating Random Primes 249
7.2.2 * Primality Testing 252
7.2.3 The Factoring Assumption 257
7.2.4 The RSA Assumption 258
7.3 Assumptions in Cyclic Groups 260
7.3.1 Cyclic Groups and Generators 260
7.3.2 The Discrete Logarithm and Diffie-Hellman Assumptions 263
7.3.3 Working in (Subgroups of) $\mathbb{Z}_p^*$ 267

7.3.4 * Elliptic Curve Groups 268
7.4 Applications of Number-Theoretic Assumptions in Cryptography 273
7.4.1 One-Way Functions and Permutations 273
7.4.2 Constructing Collision-Resistant Hash Functions . . . 276
References and Additional Reading 279
Exercises 280

8 * Factoring and Computing Discrete Logarithms 283
8.1 Algorithms for Factoring 283
8.1.1 Pollard's $p-1$ Method 284
8.1.2 Pollard's Rho Method 286
8.1.3 The Quadratic Sieve Algorithm 288
8.2 Algorithms for Computing Discrete Logarithms 291
8.2.1 The Baby-Step/Giant-Step Algorithm 293
8.2.2 The Pohlig-Hellman Algorithm 294
8.2.3 The Discrete Logarithm Problem in $\mathbb{Z}_N$ 296
8.2.4 The Index Calculus Method 297
References and Additional Reading 299
Exercises 299

9 Private-Key Management and the Public-Key Revolution 301
9.1 Limitations of Private-Key Cryptography 301
9.1.1 The Key-Management Problem 301
9.1.2 A Partial Solution – Key Distribution Centers 303
9.2 The Public-Key Revolution 306
9.3 Diffie-Hellman Key Exchange 309
References and Additional Reading 315
Exercises 315

10 Public-Key Encryption 317
10.1 Public-Key Encryption – An Overview 317
10.2 Definitions 320
10.2.1 Security against Chosen-Plaintext Attacks 322
10.2.2 Security for Multiple Encryptions 325
10.3 Hybrid Encryption 330
10.4 RSA Encryption 338
10.4.1 "Textbook RSA" and its Insecurity 338
10.4.2 Attacks on RSA 341
10.4.3 Padded RSA 344
10.5 The El Gamal Encryption Scheme 345
10.6 Chosen-Ciphertext Attacks 351
10.7 * Trapdoor Permutations and Public-Key Encryption 355
10.7.1 Trapdoor Permutations 356
10.7.2 Public-Key Encryption from Trapdoor Permutations . 356

References and Additional Reading 360
Exercises 361

11 * Additional Public-Key Encryption Schemes 363
11.1 The Goldwasser-Micali Encryption Scheme 364
11.1.1 Quadratic Residues Modulo a Prime 364
11.1.2 Quadratic Residues Modulo a Composite 366
11.1.3 The Quadratic Residuosity Assumption 370
11.1.4 The Goldwasser-Micali Encryption Scheme 371
11.2 The Rabin Encryption Scheme 374
11.2.1 Computing Modular Square Roots 375
11.2.2 A Trapdoor Permutation based on Factoring 379
11.2.3 The Rabin Encryption Scheme 383
11.3 The Paillier Encryption Scheme 385
11.3.1 The Structure of $\mathbb{Z}_{N^2}^*$ 386
11.3.2 The Paillier Encryption Scheme 388
11.3.3 Homomorphic Encryption 393
References and Additional Reading 394
Exercises 395

12 Digital Signature Schemes 399
12.1 Digital Signatures – An Overview 399
12.2 Definitions 401
12.3 RSA Signatures 404
12.3.1 "Textbook RSA" and its Insecurity 404
12.3.2 Hashed RSA 406
12.4 The "Hash-and-Sign" Paradigm 407
12.5 Lamport's One-Time Signature Scheme 409
12.6 * Signatures from Collision-Resistant Hashing 413
12.6.1 "Chain-Based" Signatures 414
12.6.2 "Tree-Based" Signatures 417
12.7 Certificates and Public-Key Infrastructures 421
References and Additional Reading 428
Exercises 429

13 Public-Key Cryptosystems in the Random Oracle Model 431
13.1 The Random Oracle Methodology 432
13.1.1 The Random Oracle Model in Detail 433
13.1.2 Is the Random Oracle Methodology Sound? 438
13.2 Public-Key Encryption in the Random Oracle Model 441
13.2.1 Security against Chosen-Plaintext Attacks 441
13.2.2 Security Against Chosen-Ciphertext Attacks 445
13.2.3 OAEP 450
13.3 RSA Signatures in the Random Oracle Model 452
References and Additional Reading 456

Exercises 457

Common Notation **459**

References **463**

A Mathematical Background **473**
A.1 Identities and Inequalities 473
A.2 Asymptotic Notation 473
A.3 Basic Probability 474
A.4 The "Birthday" Problem 476

B Supplementary Algorithmic Number Theory **479**
B.1 Integer Arithmetic 481
B.1.1 Basic Operations 481
B.1.2 The Euclidean and Extended Euclidean Algorithms . 482
B.2 Modular Arithmetic 484
B.2.1 Basic Operations 484
B.2.2 Computing Modular Inverses 485
B.2.3 Modular Exponentiation 485
B.2.4 Choosing a Random Group Element 487
B.3 * Finding a Generator of a Cyclic Group 492
B.3.1 Group-Theoretic Background 492
B.3.2 Efficient Algorithms 494
References and Additional Reading 495
Exercises 495

Part I

Introduction and Classical Cryptography

Chapter 1

Introduction and Classical Ciphers

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, including message authentication, digital signatures, protocols for exchanging secret keys, authentication protocols, electronic auctions and elections, and digital cash. 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.

现代密码学应用的
例子

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 in to the material, it will also provide an understanding of where cryptography has come from so that we can later see 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 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 often 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 (which is assumed to take place over a public channel). 窃听

We stress that 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

[1] Indeed, 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.

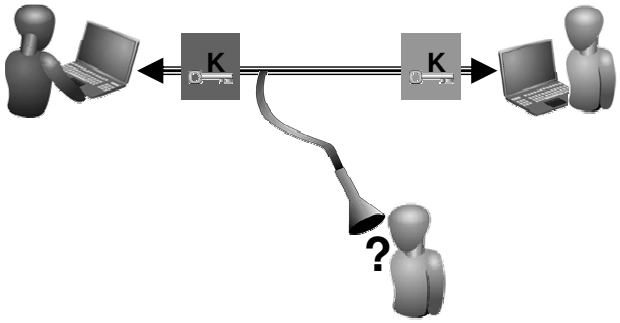

FIGURE 1.1: The basic setting of private-key encryption

in contrast to 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. We now make the above discussion a bit more formal. A private-key encryption scheme, or cipher, 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 algorithms 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 m and outputs a ciphertext c. We denote the encryption of the plaintext m using the key k by $\mathsf{Enc}_k(m)$.

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 procedure for generating keys defines a key space $\mathcal{K}$ (i.e., the set of all possible keys), and the encryption scheme is defined over some set of possible plaintext messages denoted $\mathcal{M}$ and called the *plaintext* (or *message*) *space*. Since any ciphertext is obtained by encrypting some plaintext under some key, $\mathcal{K}$ and $\mathcal{M}$ define a set of all possible ciphertexts that we denote by $\mathcal{C}$. Note that 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, an encryption scheme must have the property that decrypting a ciphertext (with 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 would compute $c := \mathsf{Enc}_k(m)$ and send the resulting ciphertext c over the public channel to the other party. 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 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 ciphers. One of the most important of these principles (known now simply as Kerckhoffs' principle) was 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 two 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 aa 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 is that in case the key *is* exposed, it is 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 (within a company, say) need to encrypt their communication, it will be significantly easier for all parties to use the same algorithm, 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 algorithm be made *public*. This stands in stark contrast with the notion of "security by obscurity" which is the idea that higher security can be achieved by keeping a cryptographic algorithm obscure (or hidden) from public view. Some of the advantages of "open cryptographic design", where the algorithm specifications are made public, include:

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 after it has been extensively studied and has withstood many attack attempts.

2. It is better that security flaws are revealed by "ethical hackers" and made public, than having the 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 effects. We stress that 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.

We remark that Kerckhoffs outlined other principles as well, and one of them states that *a system must be practically, if not mathematically, indecipherable*. As we will see later in this book, modern cryptography is based on this paradigm and — with the exception of perfectly secret encryption schemes (that are dealt with in the next chapter) — all modern cryptographic schemes can be broken in theory given enough time (say, thousands of years). Thus, these schemes are mathematically, but not practically, decipherable.

Attack scenarios. We wrap up our general discussion of encryption with a brief discussion of some basic types of attacks against encryption schemes (these will be helpful in the next section). 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 and attempts to determine the plaintext that was encrypted.
- 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 to give 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 any plaintext(s) of its choice. It then attempts to determine the plaintext that was encrypted to give 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 any ciphertext(s) of its choice. The adversary's aim, once again, is then to determine the plaintext that was encrypted to give some other ciphertext (whose decryption the adversary is unable to obtain directly).

Note that 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 types of 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 that was encrypted in some of 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 when an adversary 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 when security against these attacks is formally defined: Section 3.5 for chosen-plaintext attacks and Section 3.7 for chosen-ciphertext attacks.

We conclude by noting that different settings may require resilience 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, especially because it may be less efficient than an encryption scheme secure against "weaker" attacks; 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 **(a)** 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 **(b)** 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 in 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 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 the 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 it, 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 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 the same way, and there is no secret key. The shift cipher is similar to Caesar's cipher, but a secret key is introduced.[2] Specifically, the shift cipher uses as the key k a number between 0 and 25; to encrypt, letters are rotated (as in Caesar's cipher) but by k places. 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 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,

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

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.*[3]

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. In fact, 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 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 one-to-one in order to enable decryption. The key space thus consists of all permutations of the alphabet, meaning that the

[3] 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 when the size of the key space is equal to the size of the message space). In practice, when very long messages are typically encrypted with the same key, the key space must not be vulnerable to exhaustive search.

size of the key space is 26! (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 (or any other) language 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. Specifically, 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. However, even for quite short ciphertexts, the guesses are good enough to enable relatively quick decryption (especially utilizing knowledge of the English language, like the fact that between `t` and `e`, the character `h` is likely to appear, and the fact that `u` always 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

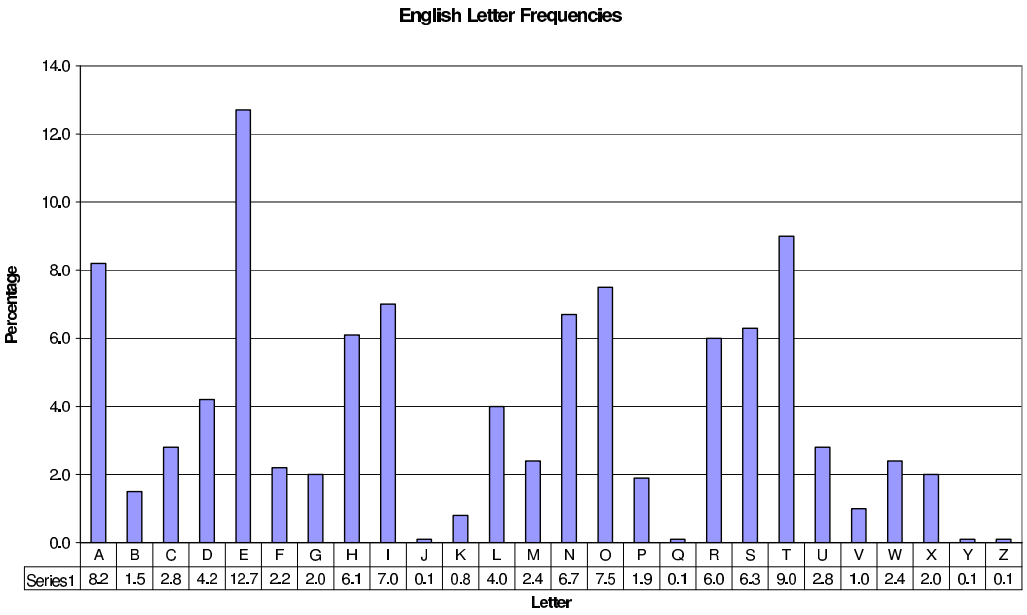

FIGURE 1.2: Average letter frequencies in the English language

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. This is another important lesson. Namely, although a large key space is *necessary* for any secure cipher, it is very far from being *sufficient*.

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.

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 the 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 again the key that is actually being used. This leads to a key-finding 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`

(Note that 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. The 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 up 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 $(1 \leq j \leq t)$ we know that 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 check which of the 26 possible keys is the correct one, for each j. This is not as trivial as in the case of the shift cipher, because by guessing a single letter of the key it is not possible to determine if the decryption "makes sense". Furthermore, checking all possible keys 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 attack method described earlier. That is, for every set of the ciphertext characters relating to a given key (that is, a given value of j), it is possible to build the frequency table of the characters and then check which of the 26 possible shifts gives 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.

One approach is to use *Kasiski's method* for solving this problem (this attack was published in the mid 19th century). The first step in the attack 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. That is, 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.

Consider the following concrete example with the password `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`

Note that 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) should be 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 (8 times the period length). Therefore, the *greatest common divisor* of all the distances between the repeated sequences should yield the period length t.

An alternate 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 $\sum_{i=0}^{25} q_i^2$ should be roughly equal to (see Equation (1.1))

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

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

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

When $\tau = t$ we expect to see $I_\tau \approx 0.065$ as discussed above. On the other hand, for $\tau \neq t$ we expect (roughly speaking) that all characters will occur roughly as often 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

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

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

Ciphertext length and cryptanalytic attacks. Notice that the above attacks on the Vigenère cipher requires 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 use for mono-alphabetic substitution also requires a longer ciphertext than for the shift cipher (which can work for messages consisting of just a single word). This phenomenon is not coincidental, and the reason for it will become more apparent after we study perfect secrecy in the next chapter.

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). An important observation is that all the above ciphers are *trivially* broken if the adversary is able to carry out a known-plaintext attack. We leave the 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 is to learn 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 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 very long time, partially due to its presumed complexity (essentially combining a number of keys together). Of course, far more complex schemes were also used, like the German Enigma. Nevertheless, this complexity does not imply security and all of these 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 this history.

1.4 The Basic Principles of Modern Cryptography

In this book, we emphasize the scientific nature of modern cryptography. In this section we will outline the main principles and paradigms that distinguish modern cryptography from the classical cryptography we studied in the previous section. 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 with 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 a definition in mind allows us to evaluate the quality of what we build and leads us toward building the right thing. 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 given an 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. Alternately, 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.[4] Alternately, there may 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.

Perhaps most importantly, precise definitions 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" and it is trivial to turn such intuition into a formal definition. For one thing, two people may each have

[4] Actually, we are simplifying a bit since things are rarely this simple.

a different intuition of what security means. Even one person might have multiple intuitive ideas of what security means, depending on the context. (In Chapter 3 we will study four different definitions of security for private-key encryption, each of which is useful in a different scenario.) Finally, it turns out that it is *not* easy, in general, to turn our intuition into a "good" definition. For example, when it comes to encryption we know that we want the encryption scheme to have the effect that only those who know the secret key can read the encrypted message. How would you formalize such a thing? The reader may want to pause to think about this before reading on.

In fact, we have asked students many times how security of 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.[5]

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 5. 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.)

3. *Answer 3 — an encryption scheme is secure if no adversary can find any of the plaintext that corresponds to the ciphertext.* This already looks like an excellent definition. However, other subtleties can arise. Going

[5] 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.

back to the example of the employment contract, it may be impossible to determine the actual salary. However, should the encryption scheme be considered secure if it were somehow possible to learn 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.

Of course, 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.)

Moreover, 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 another general principle that is 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. Regarding the break, this is exactly what we have discussed above; i.e., an encryption scheme is considered broken if an adversary can learn some function of the plaintext from a ciphertext. The *power of the adversary* relates to assumptions regarding the the actions the adversary is assumed 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., 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.) When translating this into concrete terms, we might require security against any adversary utilizes 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 what the adversary's abilities are (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 with 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. An important issue to note is that a definition of security essentially means providing a mathematical formulation of a real-world problem. If the mathematical definition does not appropriately model the real world, then the definition may be meaningless. For example, if the adversarial power that is defined 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 security needs in order for it to deliver on its mathematical promise of security.

Examples of this occur in practice all the time. As an example, an encryption scheme that has been proven secure (relative to some definition like the ones we have discussed above) might be implemented on a smart-card. It may then be possible for an adversary to monitor the power usage of the smart-card (e.g. how this power usage fluctuates over time) and use this information to determine the key. There was nothing wrong with the security definition or the proof that the scheme satisfies this definition; the problem was simply that the definition did not accurately model a 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. Alternately, 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 pervasive throughout science. To take another example from the field of computer science, consider the meaning of a *mathematical proof* that there exist well-defined problems that computers cannot solve.[6] On the one hand, such a proof is of great interest. However, the immediate question that arises is "what is a computer"? Specifically, a mathematical proof can only be provided 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 (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 that we have proven can be solved 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.*
>
> (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].*

[6] Such a proof indeed exists and it relates to the question of whether or not it is possible to check a computer program and decide whether it halts on a given input. This problem is called the Halting problem and, loosely speaking, was proven by Alan Turing to be unsolvable by computers. Those who have taken a course in Computability will be familiar with this problem and its ramifications.

In some sense, Turing faced the exact same problem as us. He developed a mathematical model of computation but needed to somehow be convinced that the model was a good one. Likewise in cryptography, we can define security and need to convinced of the fact that this implies real-world security. As with Turing, we employ the following tools to become convinced of this fact:

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 that we are familiar with are covered 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 unconditionally proven secure. This is due to the fact that their existence relies on 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 two 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 this conjecture, it is necessary for the assumption to be studied. The basic understanding is that the more the assumption is looked at 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 that one scheme is based on is *weaker* than the assumption the second scheme is based on (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 (for the reasons highlighted in the previous paragraphs).

3. *Facilitation of a proof 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 the most part) 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 scheme 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 be replaced within the higher-level constructions by another 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 show 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 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 must rely on our intuition that this is the case. Of course, intuition is in general very problematic. In fact, 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 even 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 to 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 get their software to fail. In contrast, attackers use amazingly complex and intricate means (utilizing specific properties of the construction) in order 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 that teach us that intuition in this field must not be trusted.

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 and is distinct from the ad-hoc approach that is exemplified in our study of classical cryptography and that is (unfortunately) sometimes still employed. The ad-hoc approach may fail on any one of the above three principles, but often ignores them all. Fortunately, as time goes on, a higher awareness of the necessity of a rigorous approach can be seen. Nevertheless, ad hoc implementations can still be found, especially 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 modern cryptography.

References and Additional Reading

In this chapter, we have studied just a few of the historical ciphers. There are many others of both historical and mathematical interest, and we refer to reader to textbooks by Stinson [124] or Trappe and Washington [125] 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 [79].

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 [113] 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 [70]; we will have more to say about this approach in Chapter 3.

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 i^{th} 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 basic 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 attack that works better than Kasiski's attack.

1.5 Show that the shift, substitution, and Vigenère ciphers are all trivial to break using a known-plaintext attack. (Assuming normal English text is being encrypted in each case.) How much known plaintext is needed to completely recover the key for each of the ciphers (without resorting to any statistics)?

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 early-'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 Section A.3 of Appendix A.

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

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

key space, i.e., the set of all possible keys that can be output by Gen, and 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 encryption schemes are *perfectly correct*; that is, 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. 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 that is sent is equal to m. That the message is being 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 may 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 $\Pr[M = \texttt{don't attack}] = 0.3$.

We assume that the distributions over $\mathcal{K}$ and $\mathcal{M}$ are independent, i.e., that the key and message are chosen independently. This is required because the key is chosen and fixed (i.e., shared by the communicating parties) before the message is known. Actually, recall that 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. Note that, given Enc, the distribution over $\mathcal{C}$ is fixed by the distributions over $\mathcal{K}$ and $\mathcal{M}$.

The actual 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). Then the adversary 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 are going to 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.

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 the distribution is defined by Gen and so only keys that can be output by Gen are included in the set $\mathcal{K}$ to start with.

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 both $m \in \mathcal{M}$ and $c \in \mathcal{C}$ are assigned non-zero probabilities (and thus $\Pr[M = m] > 0$ and $\Pr[C = c] > 0$, enabling us to divide by $\Pr[C = c]$ and condition on the event $M = m$). This explains our convention stated earlier, by which $\mathcal{M}$ and $\mathcal{C}$ only contain messages/ciphertexts that occur with non-zero probability.

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 over 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 $m_0, m_1 \in \mathcal{M}$ and $c \in \mathcal{C}$. By Lemma 2.2 we have that $\Pr[C = c \mid M = m_0] = \Pr[C = c]$ and $\Pr[C = c \mid M = m_1] = \Pr[C = c]$. Thus,

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

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 arbitrary $m_0 \in \mathcal{M}$ and $c \in \mathcal{C}$. Define $\gamma \stackrel{\text{def}}{=} \Pr[C = c \mid M = m_0]$. Since $\Pr[C = c \mid M = m] = \Pr[C = c \mid M = m_0] = \gamma$ for all m, we have

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

where the final equality holds for all $m \in \mathcal{M}$. So 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 an additional equivalent definition of perfect secrecy. This definition is based on an *experiment* involving an adversary $\mathcal{A}$ and its inability to distinguish the encryption of one plaintext from the encryption of another, and 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.

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 adversarial 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, the 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 (perfect secrecy — alternative definition): *An encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *over a message space $\mathcal{M}$ is* $\mathsf{perfectly\ secret}$ *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 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 (sometimes known as Vernam's cipher) 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 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 (of course, this is true as long as k is completely hidden from the adversary). We now prove this intuition formally:

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

PROOF We work directly with the original definition of perfect secrecy (Definition 2.1), though with our convention that all messages occur with non-zero probability. (For the one-time pad, this implies that all ciphertexts occur with non-zero probability.) Fix some distribution over $\mathcal{M}$ and arbitrary $m_0 \in \mathcal{M}$ and $c \in \mathcal{C}$. The key observation is that, for every $m \in \mathcal{M}$,

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

A simple calculation (using Bayes' theorem for the first equality) then gives

$$\begin{aligned}\Pr[M = m_0 \mid C = c] &= \frac{\Pr[M = m_0 \wedge C = c]}{\Pr[C = c]} \\ &= \frac{\Pr[C = c \mid M = m_0] \cdot \Pr[M = m_0]}{\sum_{m \in \mathcal{M}} \Pr[C = c \mid M = m] \cdot \Pr[M = m]} \\ &= \frac{2^{-\ell} \cdot \Pr[M = m_0]}{\sum_{m \in \mathcal{M}} 2^{-\ell} \cdot \Pr[M = m]} \\ &= \frac{\Pr[M = m_0]}{\sum_{m \in \mathcal{M}} \Pr[M = m]} \;=\; \Pr[M = m_0]\,,\end{aligned}$$

as required by Definition 2.1. ■

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.* 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

$$c \oplus c' = m \oplus m'$$

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 sufficiently-many message pairs it is possible to perform frequency analysis (as in the previous chapter, though more complex) and recover the messages themselves.

Finally, the one-time pad encryption scheme is *only secure against a ciphertext-only attack.* Although we have again not yet defined security against stronger attacks, it is easy to see that the one-time pad scheme is insecure against, e.g., a known-message attack. An adversary who obtains the encryption c of a known message m can compute the key $k = c \oplus m$ and then decrypt any subsequent ciphertexts computed using this same key.

2.3 Limitations of Perfect Secrecy

In this section, we show that one of the aforementioned limitations of the one-time pad encryption scheme is *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 limitations mentioned above are 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 $|\mathcal{K}| < |\mathcal{M}|$. Take the uniform distribution over $\mathcal{M}$ and let $m \in \mathcal{M}$ be arbitrary. Let c be a ciphertext that corresponds to a possible encryption of m; i.e., there exists a $k \in \mathcal{K}$ such that $\mathsf{Enc}_k(m) = c$. (If Enc is randomized, this means there is some non-zero probability that $\mathsf{Enc}_k(m)$ outputs c.) By correctness, we know that $\mathsf{Dec}_k(c) = m$.

Consider the set $\mathcal{M}(c)$ of all possible messages that correspond to c; that is

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

We know that $m \in \mathcal{M}(c)$. Furthermore, $|\mathcal{M}(c)| \leq |\mathcal{K}|$ 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.) This means there is some $m' \in \mathcal{M}$ with $m' \neq m$ 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 below 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* $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$. *This 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 single 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.) Once this fact is given, 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 an encryption scheme over $\mathcal{M}$ where $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$. 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 the above, $|\{\mathsf{Enc}_k(m)\}_{k\in\mathcal{K}}| \geq |\mathcal{C}|$ (because for every $c \in \mathcal{C}$ there exists a $k \in \mathcal{K}$ such that $\mathsf{Enc}_k(m) = c$). In addition, since $\mathsf{Enc}_k(m) \in \mathcal{C}$ 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 for every m and c, there do not exist distinct keys $k_1, k_2 \in \mathcal{K}$ with $\mathsf{Enc}_{k_1}(m) = \mathsf{Enc}_{k_2}(m) = c$. That is, 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 $k \in \mathcal{K}$, $\Pr[\mathcal{K} = k] = 1/|\mathcal{K}|$. 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 $k_1, \ldots, k_n$ such 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* k such that $\mathsf{Enc}_k(m_i) = c$, and furthermore these keys are distinct for distinct m_i, m_j. 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 labelling 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, and $\Pr[\mathcal{K} = k_i] = 1/|\mathcal{K}|$ 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 exists a single 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

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

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 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 ground-breaking work by Shannon [113]. In addition to introducing the notion, he proved that the one-time pad (originally introduced by Vernam [126]) is perfectly secret, and also proved the theorems characterizing perfectly-secret schemes (and their implied limitations). Stinson [124] 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. The reader interested in learning about perfectly-secure message authentication is referred to the paper by Stinson [122], the survey by Simmons [120], or the first edition of Stinson's textbook [123, 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 $c \in \mathcal{C}$:

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

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 fact 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) Describe the largest plaintext space $\mathcal{M}$ for which the mono-alphabetic substitution cipher provides perfect secrecy. (Note: this space does not need to contain words that "make sense".)

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

Reconcile the above 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 Prove that if an encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is perfectly secret for a message space $\mathcal{M}$ assuming all messages in $\mathcal{M}$ are assigned non-zero probability, then it is perfectly secret for any message space $\mathcal{M}' \subset \mathcal{M}$.

Hint: Use Shannon's theorem.

2.7 Prove the first direction of Proposition 2.5. That is, 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?

2.11 Say we require only that an encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ over a message space $\mathcal{M}$ satisfy 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 $|\mathcal{K}| < |\mathcal{M}|$.

2.12 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, then, we examine the computational approach to cryptography more generally in Section 3.1. The computational approach will be used in the rest of the book, and forms 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[1] "information" to succeed in its attack, regardless of the adversary's computational power. 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. However, 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[2] relies on assumptions whereas no assumptions are needed to achieve the latter (as we have seen in the case of encryption). 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, that 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; definitions and proofs are still essential, and it is only that we now 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 instead it 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 broken, but that *cannot* be broken with probability better than 10^{-30} in 200

[2] In theory, it is possible that these assumptions might one day be removed (though this will require, in particular, proving that $\mathcal{P} \neq \mathcal{NP}$). Unfortunately, however, our current state of knowledge requires us to make assumptions in order to prove computational security of any cryptographic construction.

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,* and
2. *Adversaries can potentially succeed with some* very small probability (*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 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, roughly speaking:

> *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 $t/2^n$. (We will see later why this is indeed optimal.) Computation on the order of $t = 2^{60}$ is barely in reach today. Running on a 1GHz computer, 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 on 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. ♢

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 100s 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.

From a theoretical standpoint, the concrete security approach is disadvantageous since 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 (e.g., 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 "efficient algorithms" with (probabilistic) algorithms running in time *polynomial in* n, meaning 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.

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 3.5). A function that grows slower than any inverse polynomial is called negligible.

We sometimes use PPT to stand for *probabilistic, polynomial-time*. 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 in practice. 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 gives a sense for 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 (that executes 10^9 cycles per second) 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*. ♢

The asymptotic approach has the advantage of not depending on any specific assumptions regarding, e.g., the type of computer an adversary will use (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.

Example 3.4
A typical proof of security for a cryptographic scheme might show that any adversary running in time $p(n)$ succeeds with probability at most $p(n)/2^n$. This implies that the scheme is (asymptotically) secure, since for *any* polynomial p, the function $p(n)/2^n$ is eventually smaller than any inverse-polynomial in n. Moreover, it immediately gives a concrete security result for any desired value of n; e.g., the scheme with n fixed to 50 is $(50^2, 50^2/2^{50})$-secure (note that for this to be meaningful we need to know the units of time with respect to which p is being measured). ♢

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 thus 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. Let us see why, somewhat informally. 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). There are two attacks, lying at opposite extremes, that apply regardless of how the encryption scheme is constructed:

- Given a ciphertext c, the 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[3] probability, in which case the adversary has found the key that the honest parties are using (and so subsequent usage of this key will 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 polynomial time and succeeds with non-zero (though very small) probability 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).

An immediate consequence of the above attacks is that asymptotic security is not possible if the key space is *fixed*, but rather the key space must depend

[3] Technically speaking, this need not be true; if it is not, however, then the scheme can be broken using a modification of this attack.

on n. That is, a private-key encryption scheme will now be associated with a sequence $\{\mathcal{K}_n\}$ such that the key is chosen from $\mathcal{K}_n$ when the security parameter is n. The above attacks imply that if we want polynomial-time adversaries to achieve only negligible success probability then the size of $\mathcal{K}_n$ must grow *super-polynomially* in the security parameter n. Otherwise, brute-force search could be carried out in polynomial time, and randomly guessing the key would succeed with non-negligible probability.

3.1.2 Efficient Algorithms and Negligible Success

In the previous section we have outlined the asymptotic security approach 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 define 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 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 overall. 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 that is 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 automatically knows the secret keys used by the honest parties.) Given this fact, 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 a *hardware random number generator* that generates 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. Note that, in either 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 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 C 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 3.5 *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 3.6
The functions $2^{-n}, 2^{-\sqrt{n}}, n^{-\log n}$ 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 $2^{-\sqrt{n}}$. $\diamondsuit$

A technical advantage of working with negligible success probabilities is that they are also closed under composition. The following is an easy exercise.

PROPOSITION 3.7 *Let* negl_1 *and* negl_2 *be negligible functions.*

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 times resulted in n heads is still negligible. (Using the union bound, 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 considered a break.*

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.) See also Example 3.1. So it simply does not make sense to worry about events that occur with sufficiently-small probability.

Asymptotic Security: A Summary

Recall that 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. The definition is always formulated so that a break that occurs with negligible probability is not considered a significant break. Thus, the general framework of any security definition is 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 (where success is also well-defined) 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 to then *prove* that the construction in question is secure given 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 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

[4] For those familiar with basic complexity theory, and in particular 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 $\mathcal{P} \neq \mathcal{NP}$, something that seems far beyond reach today.

succeeds in solving the problem that was assumed to be 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 adversary $\mathcal{A}'$ 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 execution 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) Furthermore, if $\mathcal{A}$ succeeds in "breaking" the execution 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 algorithm $\mathcal{A}$ can succeed with probability ε that is not negligible. I.e., Π is computationally secure.

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

3.2 A Definition of 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 a private-key encryption scheme; 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 3.8 *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 without loss of generality 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* $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m$*.*

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 with length parameter ℓ.

3.2.1 A Definition of Security for Encryption

There are actually a number of different ways of defining security for private-key encryption, with the main differences being with respect 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 will 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

[5] As a technical condition, note that Enc is allowed to run in time polynomial in $n + |m|$ (i.e., the total length of its inputs). We will generally only be concerned with encrypting messages of length polynomial in n, in which case this is irrelevant.

or, equivalently, is given a single ciphertext that it wishes to "crack". This is a rather weak class of adversaries (and we will encounter stronger adversaries soon), but is exactly the type of adversary that was considered in the previous chapter. Of course, as explained in the previous section, we are now interested only in adversaries that are computationally bounded to running in (probabilistic) polynomial time.

An important point to stress here is that 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 will make no assumption 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 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, and we will not present it in this book. Fortunately, there is an equivalent definition in terms of *indistinguishability* which is somewhat 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.

The definition of indistinguishability is syntactically almost identical to the alternate 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 given an encryption of one of these messages, chosen at random, using a randomly-generated key. The definition then states that a scheme Π is secure if, in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$, no adversary $\mathcal{A}$ can determine which message is encrypted with probability any different from $1/2$.

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 in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$ itself, 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 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, fixing $\mathcal{A}$ and Π, 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 (and the restriction could be removed if we were willing to forego this requirement, as we did in the case of perfect secrecy); see Exercise 3.3. This restriction, however, also turns out to be very 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.

We emphasize that most encryption schemes used in practice do *not* hide the length of messages that are encrypted. In cases when 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 should 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 adversarial 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 random key k is generated by running $\mathsf{Gen}(1^n)$, and a random bit $b \leftarrow \{0,1\}$ is chosen. The 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. If $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$, we say that $\mathcal{A}$ $\mathsf{succeeded}$.*

If Π is a fixed-length scheme with length parameter ℓ, 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 3.9 *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*

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

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 at most negligibly better than a naive guess (which is correct with probability 1/2).

An important point to notice is that the adversary is allowed to choose the messages m_0 and m_1; thus, even though it knows these plaintext messages, and knows that c is an encryption of one of them, it still (essentially) 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. 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.

An equivalent formulation. Definition 3.9 states that an eavesdropping adversary cannot detect which plaintext was encrypted with advantage significantly better than taking a random guess. An equivalent way of formalizing the definition is to state that every adversary *behaves the same way* when it receives an encryption of m_0 and when it receives 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 3.10 *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 3.9 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 to learn any partial information about the plaintext from the ciphertext. However, the actual definition of indistinguishability looks very different. As we mentioned above, Definition 3.9 is indeed equivalent to *semantic security* that formalizes the notion that partial information cannot be learned. The actual definition of semantic security is quite involved. Rather than presenting it in full and proving equivalence, we will prove two claims that demonstrate that indistinguishability implies weaker versions of semantic security. We will then present the essence of the definition of semantic security (while sweeping some details under the rug). The reader is referred to [66, Chapter 5.2] for a full definition and a full proof of equivalence.

We begin by showing that security under indistinguishability implies that no single bit of a *randomly chosen* plaintext can be guessed with probability that is significantly better than 1/2. Below, we denote by m^i the i^{th} bit of m. For technical reasons in what follows, we set $m^i = 0$ if $i > \|m\|$.

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 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 is chosen uniformly at random 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 i^{th} 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 i^{th} bit of m_0 equals 0 and the i^{th} bit of m_1 equals 1. Specifically, given a ciphertext c try to compute the i^{th} bit of the underlying plaintext. If this computation indicates that the i^{th} bit is 0, then guess that m_0 was encrypted; if it indicates that the i^{th} 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 i^{th} 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 $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with probability $1/2+\varepsilon(n)$. By the assumption that $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ has indistinguishable encryptions, $\varepsilon(\cdot)$ must be negligible.

That is, 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 i^{th} bit is 0, and let I_1^n be the set of all strings of length n whose i^{th} 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 equality holds because I_0^n and I_1^n each contain exactly half the strings of $\{0,1\}^n$. Therefore, the probability of falling in each set is exactly $1/2$.)

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

Adversary $\mathcal{A}'$:

1. On input 1^n, 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.

Note that $\mathcal{A}'$ 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

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

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 that no PPT adversary can learn any function of the plaintext, given the ciphertext. However, this is problematic to define. In particular, let us consider the function $f(m) = m^i$ (i.e., f computes the i^{th} bit). We have already shown that if m is *chosen uniformly at random*, then no adversary can predict m^i with probability better than $1/2$ given $\mathsf{Enc}_k(m)$. The first step towards generalizing this result is to show that m^i cannot be predicted for any distribution over the plaintext messages. However, if m is not uniformly distributed, then it may be possible to easily compute the i^{th} bit of m. For example, the distribution on m may be such that the i^{th} bit of m is fixed, in which case the i^{th} bit is trivial to compute. Thus, what we actually want to say is that if an adversary receiving $c = \mathsf{Enc}_k(m)$ 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, without being given the ciphertext (but only given the *a priori* distribution on m).

In the next claim we show the above when m is chosen uniformly at randomly 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 actually 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 to never 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 3.12 *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 every polynomial-time computable function* f *and every efficiently-sampleable set* 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 $m \in S_n$ uniformly at random 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 $f(m)$ with probability that is non-negligibly different depending on whether it is given an encryption of m or an encryption of 1^n, the described distinguisher violates Definition 3.10.)

The above observation yields the following algorithm $\mathcal{A}'$ that does not receive $c = \mathsf{Enc}_k(m)$, but instead receives only 1^n, yet computes $f(m)$ equally well: $\mathcal{A}'$ 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 3.12. In particular, arbitrary distributions over plaintext messages and arbitrary "external" information about the chosen plaintext are also taken into consideration. As above, we will denote by f the function of the plaintext that the adversary is attempting to compute. In addition, "external" knowledge the adversary may have regarding the plaintext is represented by a function h, and we model this "external" information by giving the adversary $h(m)$ in addition to an encryption of m. Finally, rather than considering a specific set S (and a uniform distribution over subsets of S), we consider arbitrary distributions. Specifically, we will consider a 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, so that there is 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.

DEFINITION 3.13 *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 every efficiently-sampleable distribution $X = (X_1, \ldots)$ 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.

Notice that the algorithm $\mathcal{A}$ (representing the real adversary) 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 (for example, this may represent information that is leaked about m through other means). 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 3.13 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 3.14 *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, we remark that 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 launch directly into constructions of secure encryption schemes. However, before doing so we 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. (Actually, since we are in an asymptotic setting pseudorandomness really refers to a sequence of distributions, one for each value of the security parameter. 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.

The specific types of distributions $\mathcal{D}$ we will be interested in here are those defined by choosing a short random seed $s \leftarrow \{0,1\}^n$ uniformly at random and then outputting $G(s) \in \{0,1\}^\ell$. The distribution $\mathcal{D}$ thus defined outputs the string $y \in \{0,1\}^\ell$ with probability exactly

$$\frac{\left|\{s \in \{0,1\}^n \mid G(s) = y\}\right|}{2^n}$$

which will, in general, not be the uniform distribution. Actually, we will only be interested in the case of $\ell > n$ in which case the distribution will be very far from uniform.

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 $\mathcal{D}$ as a "pseudorandom string". This is only useful shorthand, and it should be noted in particular that if $y = G(s)$ for some s then in fact y can occur as either a random or a pseudorandom string.

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 (see Section 2.2). In that case, the ciphertext is actually 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 reason that it is better to use a pseudorandom string rather than a truly random string is due to the fact 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. We now proceed to formally define the notion of a pseudorandom generator. 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. 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 3.15 *Let $\ell(\cdot)$ be a polynomial and let G be a deterministic polynomial-time algorithm such that upon 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. An important point to notice is that the output of a pseudorandom generator is actually very *far* from random. In order to see this, consider the case that $\ell(n) = 2n$ and so G doubles the length of its input. The distinguisher D receives a string of length $2n$ and must decide whether this string is truly random (i.e., uniformly distributed), or just pseudorandom. Now, 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}. Then

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

which is huge. 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 and irrespective of how they work.

The above discussion shows that the distribution generated by G is actually very far from random. Nevertheless, the key point is that *polynomial-time distinguishers* don't have time to carry out the above procedure. Furthermore, if G is indeed a pseudorandom generator, then it is guaranteed that there do not exist *any* polynomial-time procedures that succeed in distinguishing 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, and one that is clear 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. (This relies on the fact that our definitions of security are asymptotic.) In practice, however, the seed must be taken to be of some concrete length. Based on the above, 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 3.15 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, suffices it 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.

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 scheme is described in Construction 3.16, and is depicted graphically in Figure **??**.

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

CONSTRUCTION 3.16

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.

PROOF Let Π denote Construction 3.16. We show that if there exists a probabilistic polynomial-time adversary $\mathcal{A}$ for which Definition 3.9 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 3.9 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 a manner described below), and observes whether $\mathcal{A}$ succeeds or not. If $\mathcal{A}$ succeeds then D guesses that w must have been a pseudorandom string, while if $\mathcal{A}$ does not succeed then D guesses that w was a random string. In detail:

Distinguisher D:
D is given as input a string $w \in \{0,1\}^{\ell(n)}$. (We assume n can be determined from $\ell(n)$.)

1. Run $\mathcal{A}(1^n)$ to obtain the 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 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

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

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.
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 $k \leftarrow \{0,1\}^n$.

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). In contrast, when $w = G(k)$ for $k \leftarrow \{0,1\}^n$ chosen uniformly at random we have

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

(by definition of ε). Therefore,

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

where, above, w is chosen uniformly from $\{0,1\}^{\ell(n)}$ and s is chosen uniformly from $\{0,1\}^n$. Since G is a pseudorandom generator (by assumption), it must be the case that ε is negligible. Because of the way ε was defined (see Equation (3.2)), this concludes the proof that Π has indistinguishable encryptions in the presence of an eavesdropper. ■

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, Construction 3.16 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. Thus, we see that 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 3.16 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 existence of an encryption scheme satisfying Definition 3.9 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 (this is discussed further 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 might be easier than construction an encryption scheme (from scratch). (Of course, in the present case the encryption scheme does almost nothing except to XOR the output of a pseudorandom generator 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* (defined next) in Construction 3.16.

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 the security parameter is given in unary); it should then output a pseudorandom string of length ℓ. We now present the formal definition:

DEFINITION 3.18 *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 $G(s, 1^{\ell'})$.*
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.*

We remark that any standard pseudorandom generator (as in Definition 3.15) can be converted into a variable output-length one.

Given the above definition, we now modify encryption in Construction 3.16 in the natural way: encryption of a message m using the key k is done by computing the ciphertext $c := G(k, 1^{|m|}) \oplus m$; decryption of a ciphertext c using the key k is done by computing the message $m := G(k, 1^{|c|}) \oplus c$. We leave it as an exercise to prove that this scheme also has indistinguishable encryptions in the presence of an eavesdropper.

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 encrypting by 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. Using this terminology, a stream cipher is not an encryption scheme per se, but rather a *tool* for constructing encryption schemes.[6] The importance of this discussion will become more clear 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). We remark that the security

[6] 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.

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 a number of 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 (LSFRs) 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 "robust" than stream ciphers. For completeness, we remark that a stream cipher can be 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 the latter is much more efficient.

Security for Multiple Encryptions

Definition 3.9, and all our discussion until now, has dealt with the case when 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 3.9, we first introduce an appropriate experiment that is defined for an encryption scheme Π, and adversary $\mathcal{A}$, and a security parameter n:

The multi-message indistinguishability 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 $\overline{m}_0, \overline{m}_1$ such that each vector contains the same number of messages, and for all i it holds that $|m_0^i| = |m_1^i|$, where m_b^i denotes the i^{th} element of $\overline{m}_b$.*
2. *A random 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* $\overline{c}$ *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 itself remains unchanged, except that it now refers to the above experiment. That is:

DEFINITION 3.19 *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*).

An crucial point is that security for a *single* encryption (as in Definition 3.9) does not imply security under *multiple* encryptions.

CLAIM 3.20 *There exist private-key encryption schemes that are secure with respect to Definition 3.9 but that are not secure with respect to Definition 3.19.*

PROOF We do not have to look far to find an encryption scheme fulfilling the claim. Specifically, Construction 3.16, that was proven secure for a single encryption in Theorem 3.17, 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 3.16 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 $\overline{m}_0 = (0^n, 0^n)$ and $\overline{m}_1 = (0^n, 1^n)$. 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 $\overline{c} = (c^1, c^2)$ 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 3.16 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 $m_0^1 = m_0^2$); then $c^1 = c^2$ 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 3.19. ■

Necessity of probabilistic encryption. In the proof of Claim 3.20 we have shown that Construction 3.16 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 3.21 *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 satisfy Definition 3.19.*

To construct an encryption scheme secure with respect to Definition 3.19, then, we will have to (at a minimum) ensure that even when the same message is encrypted multiple times, we obtain a different ciphertext each time. (At first sight this may seem like an impossible task to achieve. However, we will see later how to achieve this.)

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 3.16) in order to encrypt multiple plaintexts. For just one example, this error appears in an implementation of encryption in Microsoft Word and Excel; see [132]. In practice, such an error can be devastating. We emphasize that this is not just a "theoretical artefact" 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.

Secure multiple encryptions using a stream cipher. There are typically two ways in which a stream cipher/pseudorandom generator can be used in practice to securely encrypt multiple plaintexts:

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. The important point to notice here is that 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 therefore follows immediately from Theorem 3.17.

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 this reason, security of this encryption method does not contradict Theorem 3.21 (as it is technical no longer an encryption scheme as per Definition 3.8.

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. The requirement, roughly speaking, 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 $G(s, IV_2)$ are pseudorandom even when viewed together and with their respective IVs. We stress that the same seed s is used each time. The above could be formalized in a similar way to Definition 3.15, by requiring that no polynomial-time distinguisher D can tell the difference between $(IV_1, G(s, IV_1), IV_2, G(s, IV_2))$ and (IV_1, r_1, IV_2, r_2) where r_1 and r_2 are independently-chosen, uniformly-distributed strings of the appropriate length.

   Given a generator as above, encryption can be defined as

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

   where IV is chosen at random. (For simplicity, we focus on encrypting fixed-length messages.) The IV is chosen fresh (i.e., 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 plaintext 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 the *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).

These two modes are depicted in Figure 3.1.

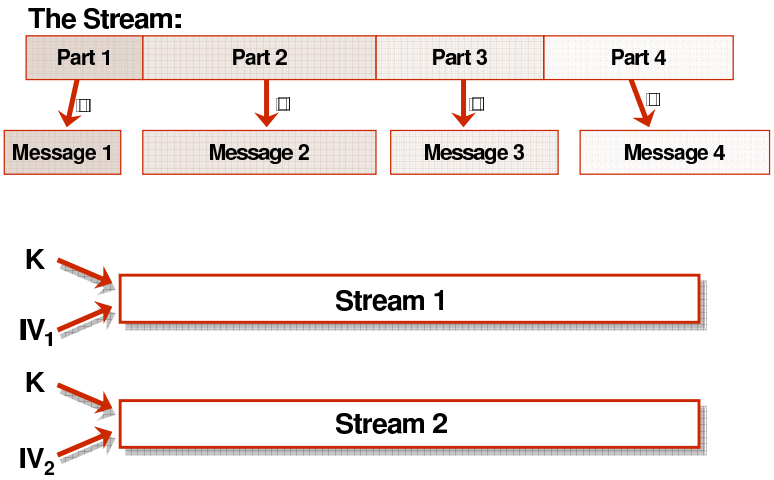

FIGURE 3.1: Synchronized mode versus unsynchronized mode

3.5 Security under 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). As compared to Definition 3.9, the definition of a break remains the same but the adversary's 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 that it chooses "on-the-fly" in an adaptive manner. 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 (these encryptions are computed using the secret key k 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 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 for the security parameter:

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

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 $\mathsf{Enc}_k(\cdot)$, 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 3.22 *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.

Before proceeding to discuss the definition, we remark that any scheme that is secure under chosen-plaintext attacks is clearly secure in the presence of an eavesdropping adversary. 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 3.22 is impossible to achieve. In particular, consider an adversary that outputs (m_0, m_1) and then receives $c \leftarrow \mathsf{Enc}_k(m_b)$. Since the adversary has oracle access to $\mathsf{Enc}_k(\cdot)$, it can request that this oracle encrypt the messages m_0 and m_1 and $\mathcal{A}$ can thus obtain $c_0 \leftarrow \mathsf{Enc}_k(m_0)$ and $c_1 \leftarrow \mathsf{Enc}_k(m_1)$. It can then compare c_0 and c_1 to its "challenge ciphertext" c; if $c = c_0$ then it 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 indeed, 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 different.[7]

Chosen-plaintext attacks in the real world. Definition 3.22 is at least as strong as our earlier Definition 3.9, 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 if it causes us to use less efficient schemes or to reject highly-efficient schemes that would suffice for "real-world applications". We should therefore ask ourselves whether chosen-plaintext attacks represent a realistic adversarial threat that we should be worried about.

In fact, chosen-plaintext attacks (in one form or another) are a realistic threat in many scenarios. We demonstrate this by first looking at 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 [87, 131] for more information.)

Coming back to the relevance of this for us here, note that the Navy cryptanalysts here carried out exactly a chosen-plaintext attack. They essentially were able to "request" (albeit in a roundabout way) the Japanese 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 they to analyze the necessity for CPA security, it is unlikely that they would have concluded that it was necessary.[8]

[7] 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 3.8, we typically consider only stateless schemes (which are preferable).

[8] It is a worthwhile mental exercise to think whether you would have anticipated such an attack.

We therefore strongly encourage always using encryption that is secure under chosen plaintext attacks.

We warn against thinking that chosen-plaintext attacks are only relevant to military applications. In fact there are many cases when 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 attackers. These attackers can therefore influence the plaintext messages that the servers encrypt, sometimes to a great extent. Such systems must therefore be protected by using an encryption scheme that is secure against chosen-plaintext attacks.

CPA security for multiple encryptions. The extension of Definition 3.22 to the case of multiple encryptions is straightforward and is the same as the extension of Definition 3.9 to Definition 3.19. That is, we define an experiment 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 Claim 3.20.) We state the claim here without proof (a similar claim, but in the public-key setting, is proved in Section 10.2.2):

CLAIM 3.23 *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 the definition of CPA security, since it suffices to prove that a scheme is secure for a single encryption and we then obtain "for free" that it is 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 has length parameter 1 so that it only encrypts messages that are 1-bit long (though everything we say extends in the natural way for any length parameter). 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 is left as an easy exercise for the reader.

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 assume that F is length-preserving so that the key, input, and output lengths of F are all the same; i.e., 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 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 function 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 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 whether it is interacting — in a sense we will more carefully define soon — 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 exactly specified by 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 $f_n \in \mathsf{Func}_n$, the look-up table for f_n 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_n 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 $n \cdot 2^n$. We conclude that the size of Func_n is $2^{n \cdot 2^n}$.

Viewing a function as a look-up table provides another useful way to think about selecting a function $f_n \in \mathsf{Func}_n$ uniformly at random. Indeed, this is exactly equivalent to choosing each row of the look-up table of f_n uniformly at random. That is, the values $f_n(x)$ and $f_n(y)$ (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_n (for $f_n \leftarrow \mathsf{Func}_n$ chosen uniformly at random). Note that 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 3.15. 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_n. 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_n); 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. (Although here the oracle is computing 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.

DEFINITION 3.24 *Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. We say F is a* pseudorandom function *if for all probabilistic polynomial-time distinguishers D, there exists a negligible function* negl *such that:*

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

where $k \leftarrow \{0,1\}^n$ is chosen uniformly at random and f_n 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 only ask 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 it is trivial to distinguish an oracle for F_k from an oracle for f_n given k: simply 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 then all claims to the pseudorandomness of F_k no longer hold. To take a concrete (though made-up) example: say F is pseudorandom. Then given oracle access to F_k (for random k), it will 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_n). But if k is known then finding such an input may be easy.

On the existence of pseudorandom functions. As with pseudorandom generators, it is important to ask whether such entities exist (and under what assumptions). For now we just note that there exist efficient primitives called *block ciphers* that are believed to act as pseudorandom functions. From a

theoretical point of view, pseudorandom functions exist if and only if pseudorandom generators exist, and so pseudorandom functions can in fact be constructed based on any of the hard problems from which pseudorandom generators can be constructed. We will discuss these issues further in Chapters 5 and 6. We remark that the existence of pseudorandom functions is very surprising, and the fact that these can be constructed based on hard problems of a certain type represents one of the 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 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 $f_n(m)$ for a random function f_n is simply a random n-bit string). However, performing encryption in this way gives a *deterministic* encryption scheme and so it cannot 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 3.25 and Figure **??**.) 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 string is used each time (since

the pseudorandom function is applied to a different input each time). Actually, this is not quite true since it is possible that a random value used for encryption repeats and is used more than once; we will have to explicitly take this into account in our proof.

CONSTRUCTION 3.25

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 3.26 *If F is a pseudorandom function, then Construction 3.25 is a fixed-length private-key encryption scheme with length parameter $\ell(n) = n$ 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_n 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 3.25, except that a truly random function f_n is used in place of F_k. That is, $\widetilde{\mathsf{Gen}}(1^n)$ chooses a random function

$f_n \leftarrow \mathsf{Func}_n$, and $\widetilde{\mathsf{Enc}}$ encrypts just like Enc except that f_n 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 adversary $\mathcal{A}$ that makes at most $q(n)$ queries to its encryption oracle, we have

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

(Note that we make no assumptions here regarding the computational power of $\mathcal{A}$.) 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 $r \in \{0,1\}^n$ is chosen and the ciphertext is set equal to $\langle r, f_n(r) \oplus m\rangle$. Let r_c denote the random string used when generating the challenge ciphertext $c = \langle r_c, f_n(r_c) \oplus m_b\rangle$. 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 $f_n(r)$ (since $f_n(r) = s \oplus m$).

   However, 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 $q(n)/2^n$.

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 $f_n(r_c)$ from its interaction with the encryption oracle (since f_n is a truly random function). That means that, as far as $\mathcal{A}$ is concerned, the value $f_n(r_c)$ that is XORed with m_b is chosen uniformly at 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

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

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 $q(\cdot)$. Note that Equation (3.4) also holds with respect to this $\mathcal{A}$. Thus, at this point, we have the following:

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

and

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

If ε is *not* negligible, then the difference between these is not negligible, either. Intuitively, such a "gap" (if present) would enable us to distinguish the pseudorandom function from a truly random function. Formally, we prove this via 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 randomly-chosen $k \leftarrow \{0,1\}^n$) or "random" (i.e., equal to f_n for randomly-chosen $f_n \leftarrow \mathsf{Func}_n$). To do this, D emulates the CPA indistinguishability experiment for $\mathcal{A}$ (in a 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 as input 1^n and has access to an oracle $\mathcal{O}$.

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 3.25. 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_n is used instead of F_k. Thus,

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

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

Since F is a pseudorandom function and D runs in probabilistic polynomial time, there exists a negligible function negl such that

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

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

$$\begin{aligned}
\mathsf{negl}(n) &\geq \left|\Pr\left[D^{F_k(\cdot)}(1^n) = 1\right] - \Pr\left[D^{f_n(\cdot)}(1^n) = 1\right]\right| \\
&= \left|\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right] - \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right]\right| \\
&\geq \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right] - \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] \\
&\geq \frac{1}{2} + \varepsilon(n) - \frac{1}{2} - \frac{q(n)}{2^n} \\
&= \varepsilon(n) - \frac{q(n)}{2^n},
\end{aligned}$$

from which we see that $\varepsilon(n) \leq \mathsf{negl}(n) + q(n)/2^n$. Since q is polynomial this means that ε is negligible, 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, the encryption of a message $m = m_1, \ldots, m_\ell$, where each m_i is an n-bit block, is given by

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

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

COROLLARY 3.27 *If F is a pseudorandom function, 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 3.25. The CPA-secure encryption scheme in Construction 3.25, 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 long plaintexts can be encrypted more efficiently.

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 (and therefore, since F is length-preserving, a bijection). We say 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.

We define what it means for a keyed permutation F to be pseudorandom in a manner analogous to Definition 3.24 but with two differences. First, we require that F_k (for a randomly-chosen k) be indistinguishable from a random *permutation* rather than a random function. This is merely an aesthetic choice since random permutations and random functions are anyway indistinguishable using polynomially-many queries. The second difference is more significant, and is motivated by the fact that cryptographic schemes using a keyed permutation may utilize the inverse F_k^{-1} in addition to F_k. Thus, we require F_k to be indistinguishable from a random permutation *even if the distinguisher is given oracle access to the inverse of the permutation.*[9] Formally:

[9] In some other works, a pseudorandom permutation is defined by considering a distinguisher that is *only* given access to the permutation (and not its inverse) as in the case of Definition 3.24, and the stronger variant we define in Definition 3.28 is called a *strong* or *super* pseudorandom permutation.

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

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

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

A pseudorandom permutation can be used in place of a pseudorandom function in any cryptographic construction. This is due to the fact that to any polynomial-time observer, a pseudorandom permutation cannot be distinguished from a pseudorandom function. Intuitively this is due to the fact that a *random* function f_n looks identical to a random permutation unless a distinct pair of values x and y are found for which $f_n(x) = f_n(y)$ (since in such a case the function cannot be a permutation). The probability of finding such points x, y using a polynomial number of queries is, however, low. We leave a proof of the following for an exercise:

PROPOSITION 3.29 *If F is a pseudorandom permutation then it is also a pseudorandom function.*

We noted earlier that a stream cipher can be modeled as a pseudorandom generator. The analogue for the case of pseudorandom permutations is a *block cipher*. Unfortunately, it is often not stated that a block cipher is actually assumed to be a pseudorandom permutation. Explicitly modeling block ciphers as pseudorandom permutations enables us to formally analyze 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.

We stress that, as with stream ciphers, block ciphers themselves are *not* secure encryption schemes. Rather, they are building blocks or tools that can be used to construct secure encryption schemes. For example, using a block cipher in Construction 3.25 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 pseudorandom permutation (block cipher) yields a scheme that is *not* CPA secure as we have already mentioned earlier.

We will study constructions of block ciphers in Chapter 5. We remark that the convention that we have 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.

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 3.27 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 having improved *ciphertext expansion* (defined to be the difference between the length of the ciphertext and the length of the message).

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 (and adding a block in case the length of the message is already a multiple of the block size). For most of the constructions in this section, we will therefore just assume that the length of the plaintext message is exactly a 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 **??** for a graphic depiction.) Decryption is done 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 3.22). Even worse, *ECB-mode encryption does not have indistinguishable encryptions in the presence of an eavesdropper, even if only used once.* 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 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.)

Mode 2 — Cipher Block Chaining (CBC) mode. In this mode, a random initial vector (IV) of length n is first chosen. Then, the first ciphertext block is generated by applying the pseudorandom permutation to $IV \oplus m_1$ (i.e., the XOR of the first plaintext block and the IV). The remainder of the ciphertext is obtained by XORing the i^{th} ciphertext block with the $i+1^{\text{th}}$ plaintext block. (See Figure **??** for a graphical depiction.) That is, set $c_0 = IV$. Then, for every $i > 0$ we set $c_i := F_k(c_{i-1} \oplus m_i)$. The final ciphertext is $\langle IV, c_1, \ldots, c_\ell\rangle$. We stress that the IV is not kept secret and is sent in

the clear as part of the ciphertext. This is crucial so that decryption can be carried out (without the IV, it will be impossible for the recipient to obtain the first plaintext block).

Importantly, encryption in CBC mode is probabilistic. Indeed, 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 is needed in order to encrypt the plaintext block m_{i+1} (unlike decryption which may be executed in parallel). Thus, if parallel processing is available, CBC-mode encryption may not be the best 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. We leave it as an exercise to show that this variant of CBC encryption is not secure.

Mode 3 — Output Feedback (OFB) mode. The third mode that 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 i^{th} block r_i of the stream equal to $r_i := F_k(r_{i-1})$. Finally, 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 **??** for a graphical depiction.) 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. This mode of operation is less common than CBC mode, but has a number of advantages. There are different variants of counter mode; we describe the *randomized counter mode*. As with OFB, counter mode can also be viewed as a way of generating a pseudorandom stream from a block cipher. First, a random $IV \in \{0,1\}^n$ is chosen; here, this IV is often denoted ctr. Then, a stream is generated by computing $r_i := F_k(\mathsf{ctr} + i)$ (where addition is performed modulo 2^n). Finally, the i^{th} plaintext block is computed as $c_i := r_i \oplus m_i$. See Figure **??** 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 i^{th} block of the ciphertext without decrypting anything else; this property is called *random access*. The above make counter mode a very attractive choice.

THEOREM 3.30 *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 3.26, 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_c denote the initial value ctr used when the challenge ciphertext is encrypted. Intuitively, when a random function f_n 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 $\mathsf{ctr}_c + i$ that was never used by the encryption oracle in answering any of its queries. This is so because if $\mathsf{ctr}_c + i$ was never used to answer a previous encryption query, then the value $f_n(\mathsf{ctr}_c + i)$ 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 $\mathsf{ctr}_c + i$ 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, a truly random function f_n is used instead. That is, $\widetilde{\mathsf{Gen}}(1^n)$ chooses a random function $f_n \leftarrow \mathsf{Func}_n$, and $\widetilde{\mathsf{Enc}}$ encrypts just like Enc except that f_n 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 would be sufficient only to require that $\mathcal{A}$ make polynomially-many queries to its encryption oracle (each query being a message of polynomial length), and output 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_c denote the initial value ctr used when the challenge ciphertext is encrypted, and let ctr_i denote the value ctr used when the encryption oracle answers the i^{th} oracle query of $\mathcal{A}$. When the challenge ciphertext is encrypted, the function f_n is applied to the value $\mathsf{ctr}_c + 1, \ldots, \mathsf{ctr}_c + \ell_c$, where $\ell_c \leq q(n)$ is the length of m_0 and m_1. When the i^{th} oracle query is answered, the function f_n 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 $j' \leq \ell_c$) for which $\mathsf{ctr}_i + j = \mathsf{ctr}_c + j'$:* In this case, the values $f_n(\mathsf{ctr}_c + 1), \ldots, f_n(\mathsf{ctr}_c + \ell_c)$ used when encrypting the challenge ciphertext are independently and uniformly distributed since f_n was not previously applied to any of these inputs. 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 $j' \leq \ell_c$) for which $\mathsf{ctr}_i + j = \mathsf{ctr}_c + j'$:* That is, the value used to encrypt block j of the i^{th} 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 learns the value of $f_n(\mathsf{ctr}_i + j) = f_n(\mathsf{ctr}_c + j')$ from the answer to its i^{th} oracle query).

Let us now analyze the probability that this occurs. The probability is maximized if ℓ_c and each ℓ_i are as large as possible, so we assume that $\ell_c = \ell_i = q(n)$ 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 $\mathsf{ctr}_c + 1, \ldots, \mathsf{ctr}_c + q(n)$, and let $\mathsf{Overlap}$ denote the event that $\mathsf{Overlap}_i$ occurs for some i. Since there are at most $q(n)$ oracle queries, we have

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

Fixing ctr_c, event $\mathsf{Overlap}_i$ occurs exactly when ctr_i satisfies

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

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 bound the success probability of $\mathcal{A}$ easily:

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

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

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

Intuitively, this is because replacing the random function f_n used in $\widetilde{\Pi}$ by the pseudorandom function F_n used in Π should have "no effect" as far as a polynomial-time adversary is concerned. Of course, this intuition should be rigorously proved; since a formal proof is very similar to the analogous step in the proof of Theorem 3.26, this is left as an exercise. ■

Block length and security. All of 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 3.26 and Theorem 3.30, 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 difference 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 3.30 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 $\frac{1}{2} + \frac{q^2}{2^{64}}$ 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.

Other modes of operation. In recent years, many different modes of operation have been introduced, offering certain advantages for certain settings.

In general, CBC, OFB, and CTR modes suffice for most applications where CPA-security is needed. Note, however, that none of these modes is secure against *chosen-ciphertext attacks*, something we will consider next.

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 we believe that the issue of *message integrity* or *message authentication* should be dealt with separately from encryption, and we do so in the next chapter. In fact, none of the above modes achieve full message integrity in the sense we will define there.

Stream ciphers versus block ciphers. As we have seen here, 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 resource-constrained devices such as PDAs or cell phones.[10] 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 our confidence in their security is lower. Furthermore, it is 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.

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 with the ability to encrypt any messages of its choice as in a chosen-plaintext attack, and also provide the adversary with the ability to *decrypt* any ciphertexts of its choice (with one exception discussed later).

[10] In particular, estimates from [45] 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.

Formally, we give the adversary access to a *decryption oracle* in addition to the 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 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* $\mathsf{Enc}_k(\cdot)$ *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 3.31 *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 3.31.

Are chosen-ciphertext attacks realistic? 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 ciphertext, 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 pattern of the 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. (Note: we do *not* recommend such a protocol, but protocols like these are sometimes suggested.) In this case, one of the honest parties may exactly act like a decryption oracle.

Insecurity of the schemes we have studied. None of the encryption schemes we have seen is CCA-secure. We will demonstrate this for Construction 3.25, 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 a "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 that we have seen yields a CCA-secure encryption scheme.

Construction of 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 [70]. 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).

Formal definitions of security against chosen-plaintext attacks were given by Luby [90] and Bellare et al. [15]. Chosen-ciphertext attacks (in the context of public-key encryption) were first formally defined by Naor-Yung [99] and Rackoff-Simon [109], and were considered also in [51] and [15]. See [83] for other notions of security for private-key encryption.

The notion of pseudorandomness was first introduced by Yao [134]. Pseudorandom generators were defined and constructed by Blum and Micali [28], who also pointed out their connection to encryption via stream ciphers (the use of stream ciphers for encryption pre-dated the formal notion of pseudorandom generators). 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 [68]. Pseudorandom permutations were studied by Luby and Rackoff [91].

Various modes of operation were standardized in [103], and the CBC and CTR modes of encryption were proven secure in [15]. 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 [93, Chapter 6]. The RC4 stream cipher is discussed in [112] and an accessible discussion of recent attacks and their ramifications can be found in [57].

Exercises

3.1 Prove Proposition 3.7.

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 3.9 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 $m_1 \in \{0,1\}^{q(n)+2}$.

3.4 Say $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is such that for k output by $\mathsf{Gen}(1^n)$, algorithm Enc_k is only defined for messages of length at most $\ell(n)$ (for some polynomial ℓ). Construct a scheme satisfying Definition 3.9 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 3.10 and Definition 3.9.

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

(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 pseudorandom functions, 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 3.19), but is not CPA-secure (i.e., is not secure with respect to Definition 3.22).

Hint: 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 key-length (i.e., $F(k,x)$ is defined only when $|x| = \log |k|$, in which case $|F(k,x)| = |k|$).

Hint: Use the fact that any random function is also pseudorandom.

3.9 Present a construction of a variable output-length pseudorandom generator from any pseudorandom function. Prove your construction secure.

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

3.11 Prove Proposition 3.29 (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 Section A.4).

3.12 Define a notion of perfect secrecy against a chosen-plaintext attack via the natural adaptation of Definition 3.22. Show that the definition cannot be achieved.

3.13 Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme defined as follows:

(a) Gen outputs a key k for a pseudorandom permutation F.

(b) Upon input $m \in \{0,1\}^{n/2}$ and key k, 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. (If you are looking for a real challenge, prove that this scheme is actually CCA-secure.)

3.14 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.

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

3.16 Complete the proof of Theorem 3.30.

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

(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 $\ell(n) \cdot \ell_{out}(n)$. For which ℓ_{in}, ℓ_{out} is the scheme CPA-secure? For which ℓ_{in}, ℓ_{out} does the scheme have indistinguishable encryptions in the presence of an eavesdropper?

3.18 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.

3.19 Show that the CBC, OFB, and counter modes of encryption do not yield CCA-secure encryption schemes.

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. One immediate question that arises, however, is what do we mean by "secure communication". In Chapter 3 we showed how 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 something about messages being sent. Specifically, when two parties communicate, they have the implicit assumption that the message sent by one party is indeed the message received by the other party. This expectation of *message integrity* is the source of a critical security concern. 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 ask itself two questions:

1. Is the order authentic? That is, did the supermarket chain really issue the order, or was it issued by an adversary who spoofed the email address of the supermarket (something that is remarkably easy to do).

2. If the order was issued by the supermarket, then the supplier must still ask whether the details of the order that it received are exactly those sent by the supermarket, or were these details somehow changed en route by an adversarial router.

Notice that the order itself is not secret and therefore the question of privacy does not arise here at all. Rather, the problem is that of *message integrity*. Such examples are very common. Indeed, any unprotected online purchase order, online banking operation, email or SMS message cannot be trusted whatsoever. Unfortunately, people are in general trusting and thus information like the ID of the caller or email return address are taken to be "proofs of identity" in many cases. This leaves the door open to potentially damaging

adversarial attacks. In this chapter we will show how to cryptographically prevent any tampering of messages that are sent over an open communication line. As we have already mentioned, the problem dealt with here is that of message authentication. We reiterate this because the goals of privacy and message authentication are often confused and unnecessarily intertwined. Having said this, at the end of this chapter, we will show how to combine encryption and authentication in a secure way so that both goals (privacy and integrity) can be simultaneously achieved.

4.2 Encryption and Message Authentication

We have already stressed that the *problems* of privacy and message authentication are distinct. However, this does not necessarily mean that their solutions are distinct. Specifically, at first sight, it may seem that encryption should immediately solve the problem of message authentication as well. This is due to the fact that a ciphertext completely hides the contents of the message. Therefore, it seems that an adversary cannot possibly modify an encrypted message en route – all that it sees is "random garbage". Despite its intuitive appeal, the claim that encryption solves the problem of message authentication is completely false.

Stream ciphers and message authentication. First, consider the case that a message m is encrypted using a stream cipher. That is, $E_k(m) = G(k) \oplus m$ where G is a pseudorandom generator. Such ciphertexts are very easy to manipulate. Specifically, flipping any bit in c results in the same bit being flipped in m upon decryption. Thus, given a ciphertext c that encrypts a message m, it is possible to modify c to c' such that $D_k(c')$ equals $D_k(c)$ except for the least significant (or any other) bit which is flipped. Note that such a modification may be very useful. For example, most electronic messages consist of a header and a body. Furthermore, headers have a fixed format and contain a number of flags. Using the strategy defined here, it is straightforward to modify flags in the header of an encrypted message (using the fixed format, an adversary can easily know which bits to flip). Needless to say, such flags can have significant meaning (for example, whether to buy or sell stock). Furthermore, if the message m represents a financial transaction where the *amount* appears in a fixed place, then an adversary can easily modify this amount. Note that even if the modification is oblivious (meaning that the adversary does not know what the amount is changed to), the result may still be very damaging.

Block ciphers and message authentication. The aforementioned attacks utilize the fact that flipping a single bit in a ciphertext generated via a stream cipher results in the flipping of the same bit in the decrypted plaintext. In

contrast, block ciphers seem to be significantly harder to attack. This is because a block cipher is a pseudorandom function and so flipping a single bit in the ciphertext of a block results in the entire block becoming scrambled upon decryption. Despite this, we argue that encryption with a block cipher still does not afford protection against message tampering. On the most basic level, one needs to assume that the recipient will be able to *detect* that one of the blocks has become scrambled. Since this may be application dependent, it cannot be relied upon as a general solution. In addition to the above, we note that the ability to tamper with a message depends on the mode of operation being used; see Section 3.6.4. Specifically, if ECB mode is used, then the order of blocks may be flipped. In this case, there is no block that has become scrambled. If OFB or CTR modes are used, then these just generate stream ciphers and so the same vulnerabilities of stream ciphers are inherent here as well. Finally, if CBC mode is used, then flipping any bit of the IV in the ciphertext results in a bit being flipped in the first block of the resulting plaintext. Since this first block may contain sensitive header information, this can yield a potentially damaging attack. We conclude that encrypting a message with a block cipher does not suffice for ensuring message integrity, even if a scrambled block can be detected (something that we argue is very problematic to assume).

As we have seen, encryption does not solve the problem of message authentication. Rather, an additional mechanism is needed that will enable communicating parties to know whether or not a message was tampered with. Such mechanisms are called message authentication codes. We remark that there is no way of preventing an adversary from modifying a message en route. The aim of a message authentication code is therefore to *detect any such modification,* so that modified messages can be discarded.

4.3 Message Authentication Codes – Definitions

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, such a task 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). The setting that we consider here therefore assumes that the parties share the same secret key. Since the parties share the same key, the notion of a message authentication code belongs to the world of *private-key cryptography.*

Loosely speaking, a message authentication code is an algorithm that is applied to a message. The output of the algorithm is a MAC tag (or just tag)

that is sent along with the message. Security is formulated by requiring that no adversary can generate a valid MAC tag on any message that was not sent by the legitimate communicating parties.

The syntax of a message authentication code. Before defining security, we first present the technical definition of what a message authentication code is. As with encryption, a message authentication code is made up of three algorithms Gen, Mac and Vrfy. The algorithm Gen generates a secret key; as with private-key encryption, we will assume that upon input 1^n, the algorithm outputs a uniformly distributed string of length n. The algorithm Mac generates MAC tags; i.e., it maps a key k and a message m to a tag t. We write this as $\mathsf{Mac}_k(m)$. Finally, the algorithm Vrfy receives a key k, a message m, and a tag t, and outputs either 1 (meaning VALID) or 0 (meaning INVALID). We write this as $\mathsf{Vrfy}_k(m,t)$. We have the following formal definition:

DEFINITION 4.1 (message authentication code – syntax): *A* message authentication code *or* MAC *is a tuple of probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *fulfilling the following:*

1. *Upon input* 1^n*, the algorithm* Gen *outputs a uniformly distributed key* k *of length* n*;* $k \leftarrow \mathsf{Gen}(1^n)$*.*
2. *The algorithm* Mac *receives for input some* $k \in \{0,1\}^n$ *and* $m \in \{0,1\}^*$*, and outputs some* $t \in \{0,1\}^*$*. The value* t *is called the* MAC tag*.*
3. *The algorithm* Vrfy *receives for input some* $k \in \{0,1\}^n$*,* $m \in \{0,1\}^*$ *and* $t \in \{0,1\}^*$*, and outputs a bit* $b \in \{0,1\}$*.*
4. *For every* n*, every* $k \in \{0,1\}^n$ *and every* $m \in \{0,1\}^*$ *it holds that* $\mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m)) = 1$*.*

If there exists a function $\ell(\cdot)$ *such that* $\mathsf{Mac}_k(\cdot)$ *is defined only over messages of length* $\ell(n)$ *and* $\mathsf{Vrfy}_k(m,t)$ *outputs* 0 *for every* m *that is not of length* $\ell(n)$*, then we say that* $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *is a* fixed length MAC *with length parameter* ℓ*.*

We remark that as for encryption, the second requirement in Definition 4.1 can be relaxed so that

$$\Pr[\mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m)) = 1] > 1 - \mathsf{negl}(n)$$

where negl is a negligible function, and the probabilities are taken over the choice of k and any internal coin tosses of Mac and Vrfy.

Security of message authentication codes. Definition 4.1 says nothing whatsoever about security. The intuitive idea behind the definition of security is that no polynomial-time adversary should be able to generate a valid MAC tag on any "new" message (i.e., a message not sent by the communicating

parties). As with any definition, we have to define the adversary's power, as well as what is considered to be a "break" of the scheme. Regarding the adversary's power, in the setting of message authentication, an adversary may be able to see many tagged messages that the parties send to each other. Furthermore, the adversary may even be able to influence the *content* of these messages. For example, consider the case that the adversary is the personal assistant of one of the parties. In this case, the adversary may be responsible for the actual wording of many of the messages that are sent (of course, we assume that the assistant does not have access to the party's key and that the party reviews any message before computing a MAC on it). Clearly, we need to ensure that this assistant is unable to generate any valid MAC on a message not reviewed by the legitimate party. In order to model the possibility that the adversary is able to effectively choose the messages that are tagged (or at least influence them to some extent), we provide the adversary with a MAC-generation box, or more technically, with a MAC oracle. During the adversarial attack, the adversary is able to request a MAC tag on any message that it wishes, where this tag is computed using the communicating parties' secret key.

At the end of this attack, the adversary attempts to break the MAC scheme. We model the break by saying that the adversary succeeds if it outputs a new message and a valid MAC tag upon that message. By a *new message*, we mean one that the adversary did not query to the MAC tag oracle. The final question that should be answered here is what form should the "new message" have. That is, should it be a valid English text, or a correctly encoded Word document? As you may have guessed, specifying any specific format would make the security of the MAC scheme dependent on a given application. Since cryptographic schemes should be secure for all applications, even those applications where random strings are tagged, we will consider a MAC scheme broken if the adversary generates *any* new message together with a valid MAC tag. This level of security is called *existential unforgeability against a chosen message attack*. The "existential unforgeability" refers to the fact that the adversary should not be able to generate a valid MAC tag on *any* message, and the "chosen message attack" refers to the fact that the adversary is able to obtain MAC tags on any messages it wishes during its attack. The formal definition of the experiment for the message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, with adversary $\mathcal{A}$ and security parameter n, is as follows:

The message authentication experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$**.**

1. *A random key* $k \leftarrow \{0,1\}^n$ *is chosen.*
2. *The adversary* $\mathcal{A}$ *is given oracle access to* $\mathsf{Mac}_k(\cdot)$ *and outputs a pair* (m,t)*. Formally,* $(m,t) \leftarrow \mathcal{A}^{\mathsf{Mac}_k(\cdot)}(1^n)$*. Let* $\mathcal{Q}$ *denote the queries asked by* $\mathcal{A}$ *during the execution.*

3. *The output of the experiment is defined to be 1 if and only if* $m \notin \mathcal{Q}$ *and* $\mathsf{Vrfy}_k(m,t) = 1$.

The definition states that no efficient adversary should 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:*

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n)$$

We remark that a message authentication code can always be broken with negligible probability (that is, there is no hope of ensuring that an adversary's success in the experiment is zero). In order to see this, let $q(\cdot)$ be a polynomial denoting the length of the MAC tags for the scheme (i.e., for a key of length n and message m, the output tag t is of length at most $q(|m|+n)$). Then, a naive attack that works for any scheme is to take any m (of any length) and simply choose a random string t of length $q(|m|+n)$. The probability that this string is a valid MAC tag is at least $2^{-q(|m|+n)}$ because at least one string constitutes a valid MAC tag. Of course, such attacks are of little consequence because their success is too small. Nevertheless, this does give us a *lower bound* on the required length of the tag in a secure MAC scheme. Specifically, the tag must be super-logarithmic; otherwise $q(|m|+n) = \mathcal{O}(\log n)$ and so $2^{-q(|m|+n)} = 2^{-\mathcal{O}(\log n)} = 1/\mathsf{poly}(n)$. In such a case, a random guess of the MAC tag is correct with non-negligible probability, and so the scheme cannot be secure.

Replay attacks and message authentication codes. Consider the following scenario: a user Alice sends her bank an order to transfer \$1,000 from her account to Bob's account. Alice is the legitimate user, and so she also applies a message authentication code to the message so that the bank knows that it is authentic. Bob is unable to intercept the message and change the sum to \$10,000 because this would involve forging the MAC scheme. However, nothing prevents Bob from intercepting Alice's message and forwarding it *ten times* repeatedly to the bank. If the bank accepts all of these messages, then \$10,000 will be transferred to Bob's account, and not \$1,000. Such an attack is called a replay attack and the MAC mechanism within itself does not prevent it. Rather, the application using the MAC is responsible for preventing replays. The reason for this is that the legitimacy or illegitimacy of replays depends on the application. Furthermore, it *cannot* be solved by considering a single isolated message; rather the context and history must be taken into account. It is thus left to the higher-level application.

Two of the possible techniques for preventing replay are using unique sequence numbers in transactions and using timestamps. When using unique sequence numbers, the idea is to not allow two transactions to have the same number (of course, this requires remembering previously used transaction numbers, but there are solutions to this as well). When using sequence numbers, the MAC is applied to the transaction content together with the sequence number. Note than any successful reply attack must forge the MAC in order to change the sequence number (if the exact same message is replayed, it will be rejected because the number has already been used). Timestamps have a similar effect: here, the message includes the current time, and some mechanism is employed so that no two transactions with the same timestamp are accepted. We stress again that the issue of replay attacks is a very real concern. However, this must be solved at the level of the *application*.

4.4 Constructing Secure Message Authentication Codes

A natural tool to use for constructing a message authentication code is a *pseudorandom function*. Intuitively, if the MAC tag t is obtained by applying a pseudorandom function to the message m, then forging a MAC involves guessing the input/output behavior of a pseudorandom function. More formally, we know that the probability of guessing the value of a random function on an unobserved point is 2^{-n} when the output length is n. It therefore follows that the probability of guessing such a value for a pseudorandom function (which is equivalent to guessing a MAC tag) can only be negligibly different.

A technicality that arises here is that our definition of pseudorandom functions (Definition 3.24) considers messages of a fixed length. Specifically, for a key of length n, the function maps inputs of length n to outputs of length n. In contrast, a MAC must be defined for all messages of all lengths. (Of course, security only holds for messages of length that is polynomial in n. However, this is implicit in the fact that a polynomial-time adversary chooses the messages and such an adversary cannot write down a message that is "too long".) We therefore first prove that a pseudorandom function constitutes a secure fixed-length MAC with length parameter $\ell(n) = n$.

We now prove that Construction 4.3 constitutes a secure MAC.

THEOREM 4.4 *Assume that the function F used in Construction 4.3 is a pseudorandom function. Then, Construction 4.3 is a fixed-length message authentication code with length parameter $\ell(n) = n$ that is existentially unforgeable under chosen message attacks.*

CONSTRUCTION 4.3 Fixed-length MAC.

Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be a function such that for every k, $F_k(\cdot)$ maps n-bit strings to n-bit strings. Define a fixed-length MAC as follows:

- $\mathsf{Gen}(1^n)$: upon input 1^n, choose $k \leftarrow \{0,1\}^n$
- $\mathsf{Mac}_k(m)$: upon input key $k \in \{0,1\}^n$ and message $m \in \{0,1\}^n$, compute $t = F_k(m)$. (If $|m| \neq |k|$ then output $\perp$.)
- $\mathsf{Vrfy}_k(m,t)$: upon input key $k \in \{0,1\}^n$, message $m \in \{0,1\}^n$ and tag $t \in \{0,1\}^n$, output 1 if and only if $t = F_k(m)$. (If the lengths are incorrect, then output 0.)

PROOF The intuition behind the proof of this theorem was described above. We therefore delve directly into 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 let $\varepsilon(\cdot)$ be a function so that

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n) \tag{4.1}$$

We show that this implies the existence of a polynomial-time algorithm that can distinguish the pseudorandom function from a random one with advantage $\varepsilon(n)$. This will then imply that ε must be negligible, as required.

Consider a message authentication code $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ which is the same as $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ in Construction 4.3 except that a truly random function f_n is used instead of the pseudorandom function F. (Of course, this is not a "legal MAC" because it is not efficient. Nevertheless, this is used for the sake of the proof only.) It is straightforward to see that

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \leq \frac{1}{2^n} \tag{4.2}$$

because for any message $m \notin \mathcal{Q}$, the value $t = f_n(m)$ 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 an oracle (that is either of a pseudorandom function or a truly random function) and works as follows. Upon input 1^n, algorithm D invokes $\mathcal{A}$ upon input 1^n. Then, when $\mathcal{A}$ queries its oracle with a message m', D queries its oracle with m' and sets t' to be the oracle reply. D hands t' to $\mathcal{A}$ and continues. At the end, when $\mathcal{A}$ outputs a pair (m,t), distinguisher D checks that m was not asked during the execution (i.e., $m \notin \mathcal{Q}$) and that t is a "valid" MAC. D does this by querying m to its oracle and checking that the response equals t. If both of the above checks pass (and so $\mathcal{A}$ "succeeded" in the experiment), then D outputs 1. Otherwise it outputs 0.

¿From the construction of D, and the success of $\mathcal{A}$ as shown in Equations (4.1) and (4.2), it follows that:

$$\Pr\left[D^{F_k(\cdot)}(1^n)=1\right] = \Pr\left[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1\right] = \varepsilon(n)$$

and

$$\Pr\left[D^{f_n(\cdot)}(1^n)=1\right] = \Pr\left[\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n)=1\right] \leq \frac{1}{2^n}$$

Therefore,

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

By the assumption that F is a pseudorandom function, it follows that $\varepsilon(n) - 2^{-n}$ must be negligible, and so $\varepsilon(\cdot)$ must be a negligible function. This implies that $\mathcal{A}$ succeeds in Mac-forge with at most negligible probability, and so Construction 4.3 is existentially unforgeable under chosen message attacks. ■

Variable-length message authentication codes. Construction 4.3 is important in that it shows a general paradigm for constructing secure message authentication codes. That is, it demonstrates that any pseudorandom function suffices. However, in its current form, this construction is only capable of dealing with messages of a fixed length; a limitation that is unacceptable in many (if not most) applications.[1] We therefore show here how a general (variable-length) MAC can be constructed from a fixed-length one. The construction here is not very efficient and is unlikely to be used in practice. Indeed, there are far more efficient constructions that have been proven secure. Nevertheless, we include this specific construction due to its simplicity. Practical constructions will be discussed later in Sections 4.5 and 4.7.

Before presenting the construction, we rule out some simple ideas. In all of the following (and in our secure construction below), the idea is to break the message into blocks and apply a pseudorandom function to the blocks in some way.

1. *Apply a pseudorandom function to the first block:* This clearly is not a secure MAC because nothing prevents an adversary from changing all the other blocks apart from the first.

[1] We note that if we had a pseudorandom function that works for variable-length inputs, then the proof of Theorem 4.4 would go through unchanged and so a variable-length MAC would be derived. However, since we did not define pseudorandom functions in this way, and since practical pseudorandom functions are for fixed input lengths, we use a different method of obtaining variable-length MACs.

2. *Exlusively-OR all of the blocks and apply a pseudorandom function to the result:* In this case, all an adversary needs to do is to change the message so that the XOR of the blocks does not change (thus implying that the MAC tag remains the same). This can be carried out by changing two of the blocks so that their XOR remains the same.

3. *Apply a pseudorandom function to each block separately and output the results:* This is similar to ECB mode in Section 3.6.4. In this case, no blocks can be easily modified. However, blocks can be removed, repeated and their order can be interchanged. The method is therefore not secure. We also note that blocks from different messages can be combined into a new message.

We leave the details of how to exactly carry out the above attacks in an effective way as an exercise.

Similarly to the above simple ideas, the actual construction (see below) works by breaking the message up into blocks and applying the pseudorandom function separately to each block. However, this must be done carefully so that the order of the blocks cannot be rearranged and so that blocks from signatures on different messages cannot be intertwined. This is achieved by including additional information in every block. Specifically, in addition to part of the message, each block contains an index of its position in the series, in order to prevent rearranging the blocks. Furthermore, all the blocks in a signature contain the same random identifier. This prevents blocks from different signatures from being combined, because they will have different identifiers. Finally, all the blocks in a signature contain the total number of blocks, so that blocks cannot be dropped from the end of the message. This brings us to the actual construction:

We now prove that Construction 4.5 constitutes a secure message authenticate code:

THEOREM 4.6 *Assume that the function F used in Construction 4.5 is a pseudorandom function. Then, Construction 4.5 is a message authentication code that is existentially unforgeable under chosen message attacks.*

PROOF The intuition behind the proof is that if the random identifier r is different in every signature that the adversary receives from the oracle, then a forgery must either contain a new identifier or it must somehow manipulate the blocks of a signed message. In both cases, the adversary must guess the output of the pseudorandom function at a "new point".

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary and let $\varepsilon(\cdot)$ be a function so that

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n) \tag{4.3}$$

CONSTRUCTION 4.5 Variable-length MAC.

Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be a function such that for every $k \in \{0,1\}^n$, $F_k(\cdot)$ maps n-bit strings to n-bit strings. Define a variable-length MAC as follows:

- $\mathsf{Gen}(1^n)$: upon input 1^n, choose $k \leftarrow \{0,1\}^n$
- $\mathsf{Mac}_k(m)$: upon input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^*$ of length at most $2^{\frac{n}{4}-1}$, first parse m into d blocks $m_1, \cdots, m_d$, each of length $n/4$. (In order to ensure unique encoding, the last block is padded with 10^*.) Next, choose a random identifier $r \leftarrow \{0,1\}^{n/4}$.

  Then, for $i = 1, \ldots, d$, compute $t_i = F_k(r\|d\|i\|m_i)$, where i and d are uniquely encoded into strings of length $n/4$ and "$\|$" denotes concatenation.[a]

  Finally, output the tag $t = (r, t_1, \ldots, t_d)$.
- $\mathsf{Vrfy}_k(m,t)$: Upon input key k, message m and tag t, run the MAC-tag generation algorithm Mac except that instead of choosing a random identifier, take the identifier r that appears in t. Output 1 if and only if the tag that is received by running Mac with this r is identical to t.

[a]Notice that i and d can be encoded in $n/4$ bits because the length of the padded m is at most $2^{n/4}$.

We show that this implies the existence of a polynomial-time algorithm that can distinguish the pseudorandom function from a random one with advantage at least $\varepsilon(n) - \mathsf{negl}(n)$ for a negligible function $\mathsf{negl}(\cdot)$. This will then imply that $\varepsilon(\cdot)$ must be negligible, as required.

Consider a message authentication code $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ which is the same as $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ in Construction 4.5 except that a truly random function f_n is used instead of the pseudorandom function F. We now show that

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \leq \mathsf{negl}(n) \tag{4.4}$$

for a negligible function $\mathsf{negl}(\cdot)$. Let $\mathcal{Q}$ be the set of queries made by $\mathcal{A}$ in $\mathsf{Mac\text{-}forge}$ with $\widetilde{\Pi}$ and let (m,t) be its output. We analyze the probability that $m \notin \mathcal{Q}$ and yet t is a valid MAC tag for m. Recall that our analysis here is in the case that a truly random function is used.

Let $t = (r, t_1, \ldots, t_d)$. We have the following cases:

1. *The identifier r appearing in the tag t output by $\mathcal{A}$ is different from all the identifiers obtained by $\mathcal{A}$ from its MAC oracle during the experiment:* This implies that the function f_n was never applied to a block of the form $(r, \star, \star, \star)$ during $\mathsf{Mac\text{-}forge}$ with $\widetilde{\Pi}$. Since f_n is truly random, it follows that the probability that $\mathcal{A}$ succeeds in guessing any single t_i is

at most 2^{-n}. (It actually needs to successfully guess all the $t_1, \ldots, t_d$ values because the new identifier r must appear in all of the blocks. Nevertheless, it suffices to bound its success by 2^{-n}.)

2. *The identifier r appearing in the tag t output by $\mathcal{A}$ appears in exactly one of the MAC tags obtained by $\mathcal{A}$ from its MAC oracle during the experiment:* Denote by m' the message that $\mathcal{A}$ queried to its oracle for which the reply t' contained the identifier r. Since $m \notin \mathcal{Q}$ it holds that $m \neq m'$, where m is the message output by $\mathcal{A}$. Let d and d' be the number of blocks in the parsing of m and m', respectively. There are two subcases here:

   (a) *Case 1: $d = d'$.* In this case, the message content of one of the blocks must be different (i.e., for some i it must hold that $(m_i, i) \neq (m'_i, i)$); let i denote the index of the different block. As in the previous case, this means that the random function f_n was never applied to a block with content (r, d, i, m_i) during Mac-forge with $\widetilde{\Pi}$, and so $\mathcal{A}$ can succeed in guessing t_i with probability at most 2^{-n}.

   (b) *Case 2: $d \neq d'$.* In this case, each block in the parsed message m is of the form $(r, d, \star, \star)$. However, r was only used in generating the MAC for m' of length d'. Therefore, f_n was never applied to a block of the form $(r, d, \star, \star)$ during the experiment. (The function f_n was only applied to blocks with a different r' or of the form $(r, d', \star, \star)$ where $d' \neq d$.) Thus, f_n was never applied to the blocks appearing in the MAC forgery t. As above, this means that $\mathcal{A}$ can succeed with probability at most 2^{-n}.

3. *The identifier r appearing in the tag t output by $\mathcal{A}$ appears in two or more of the MAC tags obtained by $\mathcal{A}$ from its MAC oracle during the experiment:* We rule out this case by showing that two MAC tags (generated legally) have the same identifier with at most negligible probability. Now, the length of a random identifier is $n/4$. Therefore, for N messages, the probability that at least two MAC tags have the same identifier is $\binom{N}{2} \cdot 2^{-n/4} = \frac{\mathcal{O}(N^2)}{2^{n/4}}$ (this calculation is based on the fact that the probability that a single pair has the same identifier is $2^{-n/4}$ and there are $\binom{N}{2}$ possible pairs). Since $N = \mathsf{poly}(n)$, we have that this is negligible, as required.

The above analysis covers all possible cases, and so we have that $\mathcal{A}$ can succeed in Mac-forge with $\widetilde{\Pi}$ with at most negligible probability, proving Equation (4.2).

The remainder of the proof follows by building a distinguisher, exactly as in the proof of Theorem 4.4. (The only difference is that the distinguisher carries out the parsing and then uses its oracle function.) Using the same

arguments, we have that

$$\left|\Pr\left[D^{F_k(\cdot)}(1^n)=1\right]-\Pr\left[D^{f_n(\cdot)}(1^n)=1\right]\right|\geq\varepsilon(n)-\mathsf{negl}(n)$$

and so $\varepsilon(\cdot)$ must be a negligible function, as required. ■

4.5 CBC-MAC

Theorem 4.6 above provides a simple proof that it is possible to construct message authentication codes for messages of any length, given only pseudorandom functions that work for a specific input/output length. Thus, for example, it demonstrates that it is possible to use *block ciphers* as a basis for constructing secure MACs. The problem, however, with the construction is that in order to compute a MAC tag on a message of length $\ell \cdot n$, it is necessary to apply the block cipher 4ℓ times. More seriously, the size of the MAC tag is $4\ell n$. Fortunately, it is possible to achieve far more efficient solutions.

The CBC-MAC construction is based on the CBC mode of encryption and is widely used in practice. It works in a similar way to Construction 4.3 in that 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 the block, the block cipher is applied ℓ times. Furthermore, the size of the MAC tag is only n bits (i.e., a single block). We begin by presenting the basic CBC-MAC construction. However, as will be discussed below, this basic scheme is *not* secure in the general case.

CONSTRUCTION 4.7 Basic CBC-MAC.

The basic CBC-MAC construction is as follows:

- $\mathsf{Gen}(1^n)$: upon input 1^n, choose a uniformly distributed string $k \leftarrow \{0,1\}^n$
- $\mathsf{Mac}_k(m)$: upon input key $k \in \{0,1\}^n$ and a message m of length $\ell \cdot n$, do the following:
  1. Denote $m = m_1, \ldots, m_\ell$ where each m_i is of length n, and set $t_0 = 0^n$.
  2. For $i = 1$ to ℓ, set $t_i \leftarrow F_k(t_{i-1} \oplus m_i)$ where $F : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ is a function.
  3. Output t_ℓ
- $\mathsf{Vrfy}_k(m,t)$: upon input key $k \in \{0,1\}^n$, a message m of length $\ell \cdot n$ and a tag t of length n, output 1 if and only if $t = \mathsf{Mac}_k(m)$

See Figure **??** for a graphical depiction of Construction 4.7. The security of this construction is given in the following theorem:

THEOREM 4.8 *Let ℓ be any fixed value. If F is a pseudorandom function such that for every $k \in \{0,1\}^n$ the function F_k maps n-bit strings to n-bit strings, then Construction 4.7 is a fixed-length MAC with length parameter $\ell \cdot n$ that is existentially unforgeable under a chosen-message attack.*

The proof of Theorem 4.8 is very involved and is therefore omitted. We stress that Construction 4.7 is only secure when the length of the messages is fixed. Of course, the advantage of this construction over Construction 4.3 is that any length can be chosen (as long as it is fixed), and we are not limited by the input/output length of the pseudorandom function. We also remark that it is not really necessary to take the length to be a multiple of n as long as padding is used.

CBC-MAC versus CBC encryption. There are two differences between the basic CBC-MAC and the CBC mode of encryption:

1. CBC encryption uses a *random IV* and this is crucial for obtaining security. In contrast, CBC-MAC uses no IV – or any fixed IV – and this is also crucial for obtaining security (i.e., a CBC-MAC with a random IV is not a secure MAC).

2. In CBC encryption all blocks are output by the encryption algorithm whereas in CBC-MAC only the last block is output. It may seem that this is a technical difference based on the fact that for 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 all blocks are output in the MAC setting, then the result is *not* a secure MAC.

In Exercise 4.4 you are asked to demonstrate attacks on a CBC-MAC that uses a random IV or one that outputs all blocks. This is a good example of the fact that harmless-looking modifications to cryptographic constructions can render them insecure. It is crucial to always implement the exact construction, and not some slight variant (unless you have a proof). Furthermore, it is crucial to understand the constructions. For example, in many cases, a cryptographic library provides a programmer with a "CBC function." However, it does not distinguish between the use of this function for encryption or for message authentication.

Construction 4.7 and variable-length messages. Theorem 4.8 states that the basic CBC-MAC construction is only secure for fixed-length messages. However, in the general case of *variable-length messages* it is easy to generate a forgery for the basic CBC construction. We leave the task of finding the

actual attack as an exercise. We remark that the attack that exists does not provide much control to the adversary over the content of the forged message. Nevertheless, as we have discussed, it is crucial that cryptographic constructions be secure for all applications and we have no way of knowing that the attack described above will not harm any application. In addition, we also have no way of knowing that other, possibly more devastating, attacks do not exist. When one attack is known, this often means that many more are possible.

Secure CBC-MAC for variable-length messages. In order to obtain a secure MAC via the CBC construction for variable-length messages, Construction 4.7 must be modified. This can be done in a number of ways. Three possible options that have been *proven secure* are:

1. Apply the pseudorandom function (block cipher) to the block length ℓ of the input message in order to obtain a key k_ℓ. Then, compute the basic CBC-MAC using the key k_ℓ and send the resulting tag along with the block length.

2. *Prepend* the message with its block length ℓ, and then compute the CBC-MAC (the first block contains the number of blocks to follow). We stress that appending the message with its block length is *not* secure.

3. Choose two different keys $k_1 \leftarrow \{0,1\}^n$ and $k_2 \leftarrow \{0,1\}^n$. Then, compute the basic CBC-MAC using k_1; let t_1 be the result. The output MAC-tag is defined to be $t = F_{k_2}(t_1)$.

We note that 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.

4.6 Collision-Resistant Hash Functions

Collision-resistant hash functions (sometimes called "cryptographic" hash functions) have many applications in cryptography and computer security. In this section we will study collision-resistant hash functions and how they are constructed. In the next section, we will show how they are used in order to construct secure message authentication codes (this explains why we study collision-resistant hash functions in this chapter).

In general, hash functions are just functions that take arbitrary-length strings and *compress* them into shorter strings. The classic use of hash func-

tions is in data structures as a way to achieve $\mathcal{O}(1)$ lookup time for retrieving an element. Specifically, if the size of the range of the hash function H is N, then a table is first allocated with N entries. Then, the element x is stored in cell $H(x)$ in the table. In order to retrieve x, it suffices to compute $H(x)$ and probe that entry in the table. Observe that since the output range of H equals the size of the table, the output length must be rather short (or else, the table will be too large). 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 may 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 their output by transforming arbitrary-length input strings into output strings of a fixed shorter length. Furthermore, as in data structures, collisions are a problem. However, there is a fundamental difference between standard hash functions and collision-resistant ones. Namely, the *desire* in data structures to have few collisions is converted into a *mandatory requirement* in cryptography. That is, a collision-resistant hash function must have the property that no polynomial-time adversary can find a collision in it. Stated differently, no polynomial-time adversary should be able to find a distinct pair of values x and x' such that $H(x) = H(x')$. We stress that in data structures some collisions may be tolerated, whereas in cryptography no collisions whatsoever are allowed. Furthermore, the adversary in cryptography specifically searches for a collision, whereas in data structures, the "data items" do not attempt to collide intentionally. This means that the requirements on hash functions in cryptography are much more stringent than the analogous requirements in data structures. It also means that cryptographic hash functions are 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. As we have mentioned, 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 (i.e., they accept all strings of all input lengths) and a finite range. Note that in such a case, collisions must *exist* (by the pigeon-hole principle). 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. However, we will only be interested in functions that compress the input, meaning that the length of the output is shorter than that of the

input. We remark that collision resistance is trivial to achieve if compression is not required: for example, the identity function is collision resistant.

More formally, one usually deals with a *family* of hash functions indexed by a "key" s. This is not a usual cryptographic key, at least in the sense that it is not kept secret. Rather, it is merely a means to specify a particular function H^s from the family. The requirement is that it must be hard to find a collision in H^s for a randomly-chosen key s. We stress that s is not a secret key and as such it is fundamentally different to the keys that we have seen so far in this book. In order to make this explicit, we use the notation H^s (rather than the more standard H_s). As usual, we begin by defining the syntax for hash functions.

DEFINITION 4.9 (hash function – syntax): *A* hash function *is a pair of probabilistic polynomial-time algorithms* (Gen, H) *fulfilling 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 included in* s *(though, we will let this be implicit).*
- *There exists a polynomial* ℓ *such that* H *is (deterministic) polynomial-time algorithm that takes as input a key* s *and any string* $x \in \{0,1\}^*$*, and outputs a string* $H^s(x) \in \{0,1\}^{\ell(n)}$*.*

If for every n *and* s*,* H^s *is defined only over inputs of length* $\ell'(n)$ *and* $\ell'(n) > \ell(n)$*, then we say that* (Gen, H) *is a* fixed-length hash function *with length parameter* ℓ'*.*

Notice that 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* the input. We remark that in the general case we have no requirement on ℓ because the function takes for input all (finite) binary strings, and so in particular all 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 begin by defining 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 chosen:* $s \leftarrow \mathsf{Gen}(1^n)$
2. *The adversary* $\mathcal{A}$ *is given* s *and outputs a pair* x *and* x'*. Formally,* $(x, x') \leftarrow \mathcal{A}(s)$*.*
3. *The output of the experiment is 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 for hash functions states that no efficient adversary can find a collision except with negligible probability.

DEFINITION 4.10 *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 (Gen, H) all using the same term "collision-resistant hash function." 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 highest level and the one we have considered so far.
2. *Second preimage resistance:* Informally speaking, a hash function is second preimage resistant if given s and x it is hard for a probabilistic polynomial-time adversary to find x' such that $H^s(x) = H^s(x')$.
3. *Preimage resistance:* Informally, a hash function is preimage resistant if given s and some y it is hard for a probabilistic polynomial-time adversary to find a value x' such that $H^s(x') = y$. This is exactly the notion of a one-way function that we will describe in Chapter 6.1 (except that here we include a key s).

Notice that any hash function that is collision resistant is second preimage resistant. Intuitively, this is the case because if given x an adversary can find 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 preimage resistant is also preimage resistant. This is due to the fact that if it is possible to invert y and find an x' such that $H^s(x') = y$ then it is possible to take x, compute $y = H^s(x)$ and invert it again obtaining x'. Since the domain of H is infinite, it follows that with good probability $x \neq x'$. We conclude that the above three security requirements form a hierarchy with each definition implying the one below it. We remark that our arguments here can all be formalized and we leave this for an exercise.

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* length-decreasing function (for simplicity of notation, we will focus on the basic case where the input domain is infinite). This attack implies a minimal output length necessary for a hash function to potentially be secure against adversaries of a certain time complexity, as is subsequently explained.

Assume that we are given a hash function $H^s : \{0,1\}^* \to \{0,1\}^\ell$ (for notational convenience we set $\ell = \ell(n)$). Then, in order to find a collision, we choose random (distinct) inputs $x_1, \ldots, x_q \in \{0,1\}^{2\ell}$, compute $y_i := H^s(x_i)$ for all 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 $q > 2^\ell$, then this occurs with probability 1. However, we are interested in the case of a smaller q. It is somewhat difficult to analyze this probability exactly, and so we will instead analyze an idealized case in which H^s is treated as a random function.[2] That is, for each x_i we assume that the value $y_i = H^s(x_i)$ is uniformly distributed in $\{0,1\}^\ell$ 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 $y_1, \ldots, y_q \in \{0,1\}^\ell$ 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*. In fact, 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 there exist two people with the same birthday? (We assume birthdays are uniformly and independently distributed among the 365 days of a non-leap year.) This is exactly the same as our problem: if y_i represents the birthday of person i, then we have $y_1, \ldots, y_q \in \{1, \ldots, 365\}$ chosen uniformly at random. Furthermore, matching birthdays correspond to distinct i, j with $y_i = y_j$ (i.e., matching birthdays correspond to collisions).

It turns out that when $q = \mathcal{O}(\sqrt{2^\ell})$ (or equivalently $\mathcal{O}(2^{\ell/2})$), then the probability of such a collision is greater than one half. (In the real birthday case, it turns out that if there are 23 people in the room, then the probability that two have the same birthday is at greater than one half.) We prove this fact in Section A.4.

Birthday attacks on hash functions – summary. If the output length of a hash function is ℓ bits then the aforementioned birthday attack finds a collision with high probability in $\mathcal{O}(q) = \mathcal{O}(2^{\ell/2})$ time (for simplicity, we assume that evaluating H^s can be done in constant time, and ignore the time

[2] Actually, it can be shown that this is (essentially) the worst case, and the algorithm finds collisions with higher probability if H^s deviates significantly from random.

required to make all the comparisons). 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. This is because if $\ell(n) = \mathcal{O}(\log n)$ then both attacks are polynomial and if $\ell(n)$ is super-logarithmic then both attacks are not polynomial. Nevertheless, in practice, birthday attacks make a huge difference. Specifically, assume that 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, 2^{64} is already feasible (albeit, still rather difficult). Thus, the existence of generic birthday attacks essentially mandates that any collision-resistant hash function in practice needs to have output that is significantly longer than 128 bits. We stress that having a long enough output is only a *necessary* condition for meeting Definition 4.10, 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 preimage or preimage resistance that are quicker than time 2^ℓ.

Improved birthday attacks. The birthday attack that we described above has two weaknesses. First, it requires a large amount of memory. Second, we have very little control over the colliding values (note that although we choose the q values $x_1, \ldots, x_q$ we have no control over which x_i and x_j are likely to collide). It is possible to construct better birthday attacks as follows.

The basic birthday attack requires the attacker to store all q values, because it cannot know which pair will form a collision until it happens. This is very significant because, for example, it is far more feasible to run in 2^{64} time than it is to obtain a disk of size 2^{64}. Nevertheless, it is possible to construct a birthday attack that takes time $2^{\ell/2}$ as above, yet requires only a constant amount of memory. We will only describe the basic idea here, and leave the details as an exercise. The idea is to take two random values x_1 and y_1 and then for every $i > 1$ to compute $x_i = H(x_{i-1})$ and $y_i = H(H(y_{i-1}))$, until we have some m for which $x_m = y_m$. Notice that this means that $H(x_{m-1}) = H(H(y_{m-1}))$ and so x_{m-1} and $H(y_{m-1})$ constitute a collision in H (as long as they are distinct, which holds with good probability). It can be shown that this collision is expected to appear after $\mathcal{O}(2^{\ell/2})$ steps, as with the regular birthday attack. Thus, it is not necessary to use a large amount of memory in order to carry out a birthday attack.

The second weakness that we mentioned relates to the lack of control over the colliding messages that are found. We stress that it is not necessary to find "nice" collisions in order to break cryptographic applications that use collision-resistant hash functions. Nevertheless, it is informative to see that birthday attacks can be made to work on messages of a certain form. Assume that an attacker wishes to prepare two messages x and x' such that $H(x) = H(x')$. Furthermore, the first message x is a letter from her employer that she was

fired for lack of motivation at work, while the second message x' is a flattering letter of recommendation from her employer. Now, a birthday attack works by generating $2^{\ell/2}$ different messages and it seems hard to conceive that this can be done for the aforementioned letters. However, it is actually possible to write the same sentence in many different ways. For example, consider the following sentence:

> It is *(hard)(difficult)(infeasible)* to *(find)(obtain)(acquire)(locate)* collisions in *(cryptographic)(collision-resistant)* hash functions in *(reasonable)(acceptable) (unexcessive)* time.

The important point to notice about this sentence is that any combination of the words is possible. Thus, the sentence can be written in $3 \cdot 4 \cdot 2 \cdot 3 > 2^6$ 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 to prepare $2^{\ell/2}$ letters explaining why the attacker was fired and another $2^{\ell/2}$ letters of recommendation and with good probability, a collision between the two types of letters will be found. We remark that 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 that is widely used for constructing collision-resistant hash functions. The task of constructing a collision-resistant hash function is difficult. It is made even more difficult by the fact that the input domain must be infinite (any string of any length may be input). The Merkle-Damgård transform is a way of extending a *fixed-length* collision-resistant hash function into a general one that receives inputs of any length. The method works for any fixed-length collision-resistant hash function, even one that reduces the length of its input by just a single bit. This transform therefore reduces the problem of designing a collision-resistant hash function to the (easier) problem of designing a fixed-length collision-resistant function that compresses its input by any amount (even a single bit). The Merkle-Damgård transform is used in practical constructions of hash functions, and is also very interesting from a theoretical point of view since it implies that compressing by one bit is as easy (or as hard) as compressing by an arbitrary amount.

We remark that in order to obtain a full construction of a collision-resistant hash function it is necessary to first construct a fixed-length collision-resistant hash function. We will not show how this is achieved in practice. However, in Section 7.4.2 we will present theoretical constructions of fixed-length collision-resistant hash functions. These constructions constitute a proof that it is possible to obtain collision-resistant hash functions under standard cryptographic assumptions.

For concreteness, we look at 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 4.7 you are asked to generalize the construction for any $\ell' > \ell$. We denote the given fixed-length collision-resistant hash function by (Gen_h, h) and use it to construct a general collision-resistant hash function (Gen, H) that maps inputs of any length to outputs of length $\ell(n)$. We remark that in much of the literature, the fixed-length collision-resistant hash function used in the Merkle-Damgård transform is called a *compression function*. The Merkle-Damgård transform is defined in Construction 4.11 and depicted in Figure **??**.

CONSTRUCTION 4.11 The Merkle-Damgård Transform.

Let (Gen_h, h) be a fixed-length hash function with input length $2\ell(n)$ and output length $\ell(n)$. Construct a variable-length hash function (Gen, H) as follows:

- $\mathsf{Gen}(1^n)$: upon input 1^n, run the key-generation algorithm Gen_h of the fixed-length hash function and output the key. That is, output $s \leftarrow \mathsf{Gen}_h$.
- $H^s(x)$: Upon input key s and message $x \in \{0,1\}^*$ of length at most $2^{\ell(n)} - 1$, compute as follows:
  1. Let $L = |x|$ (the length of x) and let $B = \lceil \frac{L}{\ell} \rceil$ (i.e., the number of blocks in x). Pad x with zeroes so that its length is an exact multiple of ℓ.
  2. Define $z_0 := 0^\ell$ and then for every $i = 1, \ldots, B$, compute $z_i := h^s(z_{i-1} \| x_i)$, where h^s is the given fixed-length hash function.
  3. Output $z = H^s(z_B \| L)$

We remark that we limit the length of x to be at most $2^{\ell(n)} - 1$ so that its length can fit into a single block of length $\ell(n)$ bits. Of course, this is not a limitation because we assume that all messages considered are of length polynomial in n and not exponential.

The initialization vector. We remark that the value z_0 used in step 2 is arbitrary can be replaced by any constant. This value is typically called the IV or *initialization vector*.

THEOREM 4.12 *If (Gen_h, h) is a fixed-length collision-resistant hash function, then (Gen, H) is a collision-resistant hash function.*

PROOF We first 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 $x_1 \cdots x_B$ be the B blocks of the padded x, and let $x'_1 \cdots x'_{B'}$ be the B' blocks of the padded x'. 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 $z = h^s(z_B \| L)$ and of $H^s(x')$ is $z = h^s(z_{B'} \| L')$. 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 $h_B \| L$ and $h_{B'} \| L'$ are two different strings that collide for h^s.

2. *Case 2 – $L = L'$:* Let z_i and z'_i be the intermediate hash values of x and x' (as in Figure **??**) during the computation of $H^s(x)$ and $H^s(x')$, respectively. Since $x \neq x'$ and they are of the same length, there must exist at least one index i (with $1 \leq i \leq B$) such that $x_i \neq x'_i$. Let i^* be the *highest* index for which it holds that $z_{i^*-1} \| x_{i^*} \neq z'_{i^*-1} \| x'_{i^*}$. If $i^* = B$ then $(z_{i^*-1} \| x_{i^*})$ and $(z'_{i^*-1} \| x'_{i^*})$ constitutes a collision because we know that $H^s(x) = H^s(x')$ and $L = L'$ implying that that $z_B = z'_B$. If $i^* < B$, then the maximality of i^* implies that $z_{i^*} = z'_{i^*}$. Thus, once again, $(z_{i^*-1} \| x_{i^*})$ and $(z'_{i^*-1} \| x'_{i^*})$ constitutes a collision. That is, in both cases, we obtain that

$$z_{i^*-1} \| x_{i^*} \neq z'_{i^*-1} \| x'_{i^*}$$

while

$$h^s(z_{i^*-1} \| x_{i^*}) = h^s(z'_{i^*-1} \| x'_{i^*}),$$

meaning that we have found a collision in 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

In Section 7.4.2 we will present theoretical constructions of collision-resistant hash functions whose security can be reduced to standard cryptographic assumptions. Unfortunately, those constructions are rather inefficient and so are not used in practice. Rather, the constructions in practice are heuristic in nature and we do not present them here. Nevertheless, we present a few important remarks about these functions.

One important difference between collision-resistant hash functions used in practice and the notion we discussed above is that the hash functions in practice are generally *unkeyed*, rather than keyed. Essentially this means that a fixed hash function H is defined and there is no longer any notion of a Gen algorithm generating a key s for H. The Gen algorithm was included in the formal definition because the known theoretical constructions all rely in an

essential way on the use of a key. In fact, subtle technical difficulties arise even in trying to define a meaningful notion of collision resistance for unkeyed hash functions.[3] On a pragmatic level, once a collision is 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 way. On the other hand, if H were a keyed function then a collision for H^s does not necessarily make it any easier to find a collision in $H^{s'}$ for a freshly-generated key s' (although an algorithm that *finds* collisions may succeed for all keys).

An even more fundamental technical difference arises if H has a fixed output length, which is the case for most practical hash function constructions: in this case, there is no longer any notion of a security parameter and so it no longer makes any sense to speak of a "polynomial-time algorithm" finding collisions with "negligible probability." (Specifically, relying on the birthday bounds, a collision in a hash function with constant output length ℓ can always be found with high probability in constant(!) time $2^{\ell/2}$.) Instead, the most that can be claimed is that it is infeasible for any algorithm running in some "reasonable" amount of time to find a collision in H with "significant" probability.

Nevertheless, unkeyed hash functions are used extensively in practice. The reason is that they are vastly more efficient than the known theoretical constructions and, in practice, the security guarantee stated in the previous paragraph is enough. In particular, if ℓ is large enough the possibility of finding a collision in constant time $2^{\ell/2}$ is not a concern. Indeed, 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} which is out of reach today. We remark that despite the above, it is possible to reconcile the unkeyed hash functions in practice with the keyed hash functions in theory. We present two such reconciliations now:

1. Collision-resistant hash functions in practice have a fixed initialization vector IV (as in Merkle-Damgård) as part of the code. One could argue that the IV is essentially the key s. Similarly, the code of collision-resistant hash functions in practice typically includes certain constants. Again, these constants can be viewed as part of the key s. Note that viewed in this way, the key s was chosen once and for all. Nevertheless, also in theory, once s is chosen it can be used by everyone and for many years.

2. The proofs of security that rely on collision resistance all show that if the considered construction is not secure, then a collision can be found in the hash function (as with the Merkle-Damgård transform). Thus,

[3] To get a sense for the technical problem, let x, x' be a collision for a fixed hash function H (if H is length decreasing, 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.

when considering unkeyed functions, this could be translated into saying that "if a real-world adversary breaks the considered construction, then it is possible to construct a real-world algorithm that finds an actual collision in the hash function". If we believe that it is hard to find an actual collision in the hash functions used in practice, then this gives us a good security guarantee.

Two popular hash functions are MD5 and SHA-1. 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). The Merkle-Damgård transform (or actually something very similar) is then applied to the compression function in order to obtain a collision-resistant hash function for arbitrary-length messages. 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" (cf. Section 4.6.3) 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 method of attacking hash functions. The attack worked on MD5 and a large number of other hash functions. In addition to a general algorithm (that finds collisions for every IV), the team presented an actual collision in MD5. (Prior to this, weaknesses were known to exist in MD5, but no full collision was ever found.) The algorithm is such that there is 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 two postscript files containing whatever content is desired. A year later, the Chinese team applied the new approach to SHA-1 and demonstrated an algorithm for finding collisions in SHA-1 using less time than that required by a generic birthday attack. The attack on SHA-1 requires time 2^{69} and as of yet, no explicit collision has been found. These attacks have motivated a gradual shift toward hash functions with larger outputs lengths which are presumed to be 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 these results is that there is great interest today in developing new hash functions and a new hash standard. We stress that today MD5 has been *completely broken* and collisions can be found in mere minutes. Thus, MD5 should not be used in any application where collision-resistance is required (and it is prudent not to use it even when only second preimage resistance is needed).

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 completely different approach is taken in the NMAC and HMAC constructions which are based on collision-resistant hash functions. Loosely speaking (and not being entirely accurate), the constructions of NMAC and HMAC rely on the following assumptions regarding the collision-resistant hash function being used:

1. The hash function is constructed using the Merkle-Damgård transform, and
2. The fixed-length collision-resistant hash function – typically called a *compression function* – that lies at the heart of the hash function, has certain pseudorandom or MAC-like properties.

The first assumption is true of most known collision-resistant hash functions. The second assumption is believed to be true of the hash functions that are used in practice and believed to be collision resistant. We remark that the security of NMAC and HMAC actually rests on a weaker assumption than that described above, as we will discuss below.

We first present NMAC and then HMAC, since as we will see, HMAC can be cast as a special case of NMAC. Furthermore, it is easier to first analyze the security of NMAC and then derive the security of HMAC.

Notation – the IV in Merkle-Damgård. In this section we will explicitly refer to the IV used in the Merkle-Damgård transform of Construction 4.11; recall that this is the value given to z_0. In the standard Merkle-Damgård construction, the IV is fixed. However, here we will wish to vary it. We denote by $H^s_{IV}(x)$ the computation of Construction 4.11 on input x, with key s and z_0 set to the value $IV \in \{0,1\}^\ell$.

4.7.1 Nested MAC (NMAC)

Let H be a hash function of the Merkle-Damgård type, and let h be the compression function used inside H. For simplicity, we denote the output length of H and h by n (rather than by $\ell(n)$). The first step in the construction of NMAC is to consider *secretly keyed* versions of the compression and hash functions. This is achieved via the initialization vector IV. Specifically, for a given hash function H^s constructed according to Merkle-Damgård and using (the non-secret) key s, define H^s_k to be the keyed hash function obtained by setting $IV = k$, where k is secret. Likewise, for a compression function h^s, define a secretly keyed version by $h^s_k(x) = h^s(k\|x)$. That is, the keyed compression function works by applying the unkeyed compression function to the concatenation of the key and the message. Note that in the

Merkle-Damgård construction, the use of the IV is such that the first iteration computes $z_1 = h^s(IV \| x_1)$. Here $IV = k$ and so we obtain that $z_1 = h^s(k\|x)$ which is the same as $h^s_k(x)$. This implies that the secret key for H and h can be of the same size. By the construction, this equals the size of the output of h, and so the keys are of length n. To summarize, we define secretly keyed compression and hash functions as follows:

- For a compression function h with non-secret key s, we define $h^s_k(x) \stackrel{\text{def}}{=} h^s(k\|x)$ for a secret key k.
- For a Merkle-Damgård hash function H with non-secret key s, we define $H^s_k(x)$ to equal the output of H^s on input x, when the IV is set to the secret key k. This is consistent with notation above of $H^s_{IV}(x)$.

We are now ready to define the NMAC function:

CONSTRUCTION 4.13 Nested MAC (NMAC).

The NMAC construction is as follows:

- $\mathsf{Gen}(1^n)$: upon input 1^n, run the key-generation for the hash function obtaining s, and choose $k_1, k_2 \leftarrow \{0,1\}^n$.
- $\mathsf{Mac}_k(m)$: upon input (s, k_1, k_2) and $x \in \{0,1\}^*$, compute $NMAC^s_{k_1,k_2}(x) = h^s_{k_1}(H^s_{k_2}(x))$
- $\mathsf{Vrfy}_k(m,t)$: Output 1 if and only if $t = \mathsf{Mac}_k(m)$

In words, NMAC works by first applying a keyed collision-resistant hash function H^s to the input x, and then applying a keyed compression function to the result. Notice that the input/output sizes are all appropriate (you should convince yourself of this fact). $H^s_{k_2}$ is called the *inner* function and $h^s_{k_1}$ the *outer* function.

The security of NMAC relies on the assumption that H^s_k with a secret k is collision-resistant, and that h^s_k with a secret k constitutes a secure MAC. In order to state the security claim formally, we define the following:

- Given a hash function (Gen, H) generated via the Merkle-Damgård transform, define $(\hat{\mathsf{Gen}}, \hat{H})$ to be a modification where $\hat{\mathsf{Gen}}$ runs Gen to obtain s and in addition chooses $k \leftarrow \{0,1\}^n$. Furthermore, $\hat{H}$ is computed according to Construction 4.11 using $IV = k$ (that is, $\hat{H}_{s,k}(x) = H^s_k(x)$).
- Given a fixed-length hash function (Gen, h) mapping $2n$ bits to n bits, define $(\hat{\mathsf{Gen}}, \hat{h})$ to be a modification where $\hat{\mathsf{Gen}}$ runs Gen to obtain s and in addition chooses $k \leftarrow \{0,1\}^n$. Furthermore, $\hat{h}_{s,k}(x) = h^s(k\|x)$. Define a message authentication code based on $(\hat{\mathsf{Gen}}, \hat{h})$ by computing $\mathsf{Mac}_{s,k}(x) = \hat{h}_{s,k}(x)$ and Vrfy in the natural way.

We have the following theorem:

THEOREM 4.14 *Let* (Gen, h) *be a fixed-length hash function and let* (Gen, H) *be a hash function derived from it using the Merkle-Damgård transform. Assume that the secretly-keyed* $(\hat{\mathsf{Gen}}, \hat{H})$ *defined as above is collision-resistant, and that the secretly keyed* $(\hat{\mathsf{Gen}}, \hat{h})$ *as defined above is a secure fixed-length message authentication code. Then, NMAC described in Construction 4.13 is a secure* (*arbitrary-length*) *message authentication code.*

We will not present a formal proof of this theorem here. Rather, an almost identical proof is given later in Section 12.4 in the context of digital signatures (see Theorem 12.5). It is a straightforward exercise to translate that proof into one that works here as well.

For now, we will be content to sketch the idea behind the proof of Theorem 4.14. Assume, by contradiction, that a polynomial-time adversary $\mathcal{A}$ manages to forge a MAC. Recall that $\mathcal{A}$ is given an oracle and can ask for a MAC on any message it wishes. Then, it is considered to successfully forge if it outputs a valid MAC tag on any message that it did not ask its oracle. Let m^* denote the message for which $\mathcal{A}$ produces a forgery and let $\mathcal{Q}$ denote the set of queries it made to its oracle (i.e., the set of messages that for which it obtained a MAC tag). 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 equals the MAC tag for m^* and so clearly $\mathcal{A}$ can successfully forge. However, this case directly contradicts the assumption that H_k is collision resistant because $\mathcal{A}$ found distinct m and m^* for which $H^s_{k_2}(m^*) = H^s_{k_2}(m)$.

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 the message m^* for which the MAC forgery is formed is such that $H^s_{k_2}(m^*) \notin \mathcal{Q}'$. Thus, we can consider the attack by $\mathcal{A}$ to be a MAC attack on $h^s_{k_1}$ where the messages are all formed as $H^s_{k_2}(m)$ for some m, and the message for which the forgery is generated is $H^s_{k_2}(m^*)$. In this light, we have that $\mathcal{A}$ successfully generates a MAC forgery in the *fixed-length* message authentication code h^s_k. This contradicts the assumption that h^s_k is a secure MAC.

Once again, for a formal proof of this fact, see the proof of Theorem 12.5 (the only modifications necessary are to replace "signature scheme" with "message authentication code").

Security assumptions. We remark that if the inner hash function is assumed to be collision resistant (in the standard sense), then no secret key k_2 is needed. The reason why NMAC (and HMAC below) were designed with secret keying for the inner function is that it allows us to make a weaker

assumption on the security of H^s. Specifically, even if it is possible to find collisions in H^s, it does not mean that it is possible to find collisions in the *secretly* keyed version H^s_k. Furthermore, even if it is possible to find collisions, notice that the attacker cannot compute millions of hashes by itself but must query its MAC oracle (in real life, this means that it must obtain these values from the legitimately communicating parties). To make its life even harder, the attacker 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 $H^s_{k_2}(x)$.

Despite the above, it is not clear that the assumption that H^s_k is collision resistant is significantly "better" than the assumption that H^s is collision resistant. For example, the new attacks on MD5 and other hash functions that we mentioned above work for *any* IV. Furthermore, the prefix of any message in a Merkle-Damgård hash function can essentially be looked at as an IV. Thus, if it is possible to obtain a single value $H^s_k(x)$ for some x, then one can define $IV = H^s_k(x)$. Then, a collision in H^s_{IV} can be used to derive a collision in H^s_k, even though a secret key k is used. We reiterate that in NMAC, the adversary actually only receives $h^s_{k_1}(H^s_{k_2}(x))$ and not $H^s_{k_2}(x)$. Nevertheless, the statement of security in Theorem 4.14 does not use this fact (and currently it is not known how to utilize it).

4.7.2 HMAC

The only disadvantage of NMAC is that the IV of the underlying hash function must be modified. In practice this can be annoying because the IV is fixed by the function specification and so existing cryptographic libraries do not enable an external IV input. Essentially, HMAC solves this problem by keying the compression and hash functions by concatenating the input to the key. Thus, the specification of HMAC refers to an unkeyed hash function H. Another difference is that HMAC uses a single secret key, rather than two secret keys.

As in NMAC, let H be a hash function of the Merkle-Damgård type, and let h be the compression function used inside H. For the sake of concreteness here, we will assume that h compresses its input by exactly one half and so its input is of length $2n$ (the description of HMAC depends on this length and so must be modified for the general case; we leave this for an exercise). The HMAC construction uses two (rather arbitrarily chosen) fixed 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. The string opad is formed by repeating the byte 36 in hexadecimal as many times as needed; the string ipad is formed in the same way using the byte 5C. The HMAC construction, with an arbitrary fixed IV, is as follows (recall that '$\|$' denotes concatenation):

We stress that "$k \oplus \mathsf{ipad} \,\|\, x$" means that k is exclusively-ored with ipad and the result is concatenated with x. Construction 4.15 looks very different from Construction 4.13. However, as we will see, it is possible to view HMAC as a special case of NMAC. First, assume that h^s_k with a secret k behaves like

CONSTRUCTION 4.15 HMAC.

The HMAC construction is as follows:

- $\mathsf{Gen}(1^n)$: upon input 1^n, run the key-generation for the hash function obtaining s, and choose $k \leftarrow \{0,1\}^n$.
- $\mathsf{Mac}_k(m)$: upon input (s,k) and $x \in \{0,1\}^*$, compute

$$HMAC^s_k(x) = H^s_{IV}\Big(k \oplus \mathsf{opad} \parallel H_{IV}\Big(k \oplus \mathsf{ipad} \parallel x\Big)\Big)$$

  and output the result.
- $\mathsf{Vrfy}_k(m,t)$: output 1 if and only if $t = \mathsf{Mac}_k(m)$.

a pseudorandom function (note that SHA-1 is often used as a pseudorandom generator in practice and so this is believed to be the case). Now, the first step in the computation of the inner hash $H^s_{IV}(k \oplus \mathsf{ipad} \parallel x)$ is $z_1 = h^s(IV \parallel k \oplus \mathsf{ipad})$. Likewise, the first step in the computation of the outer hash is $z'_1 = h^s(IV \parallel k \oplus \mathsf{opad})$. Looking at h as a pseudorandom function, this implies that $h^s(IV \parallel k \oplus \mathsf{ipad})$ and $h^s(IV \parallel k \oplus \mathsf{opad})$ are essential different pseudorandom keys k_1 and k_2. Thus, denoting $k_1 = h^s(IV \parallel k \oplus \mathsf{ipad})$ and $k_2 = h^s(IV \parallel k \oplus \mathsf{opad})$ we have

$$HMAC^s_k(x) = H^s_{IV}\big(k \oplus \mathsf{opad} \parallel H\big(k \oplus \mathsf{ipad} \parallel x\big)\big) = H^s_{k_1}(H^s_{k_2}(x)).$$

Notice now that the outer application of H is to the output of $H^s_{k_2}(x)$ and so is just a single computation of h^s. We therefore conclude that

$$HMAC_k(x) = h^s_{k_1}(H^s_{k_2}(x))$$

which is exactly NMAC. In order to formally restate the theorem of NMAC for HMAC, we just need to assume that the key derivation of k_1 and k_2 using ipad and opad yields two pseudorandom keys, and everything else remains the same. We formally state this by defining the function G by

$$G(k) = h^s(IV \| k \oplus \mathsf{opad}) \parallel h^s(IV \| k \oplus \mathsf{ipad})$$

and requiring that it be a pseudorandom generator with expansion factor $2n$ (see Definition 3.15 in Chapter 3). Observe that the generator doubles its length and so produces two pseudorandom keys each of length n from a single random key of length n. Keying the hash functions as we have described above, we have the following:

THEOREM 4.16 *Assume that* (Gen, H), $(\hat{\mathsf{Gen}}, \hat{H})$, (Gen, h) *and* $(\hat{\mathsf{Gen}}, \hat{h})$ *are all as in Theorem 4.14. In addition, assume that* G *as defined above is a pseudorandom generator. Then, HMAC described in Construction 4.15 is a secure (arbitrary-length) message authentication code.*

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 all hash functions in practice that are considered collision resistant). The importance of HMAC is partially due to the timeliness of its appearance. When HMAC was presented, many practitioners refused to use CBC-MAC (with the claim that is is "too slow") and instead used heuristic constructions that were often insecure. For example, a MAC was defined as $H(k\|x)$ where H is a collision-resistant hash function. It is not difficult to show that when H is constructed using Merkle-Damgård, this is not a secure MAC at all.

4.8 * Achieving Chosen-Ciphertext Secure Encryption

In Section 3.7, we introduced the notion of CCA security. In this section we will use message authentication codes (and CPA secure encryption) in order to construct a CCA-secure private-key encryption scheme.

Constructing CCA-secure encryption schemes. In order to achieve CCA-security, we will construct an encryption scheme with the property that the adversary will not be able to obtain *any* valid ciphertext that was not generated by the legitimate parties. This will have the effect that the decryption oracle will be rendered useless. Given this intuition, it is clear why message authentication codes help. Namely, our construction works by first encrypting the plaintext message, and then applying a MAC to the resulting ciphertext. This means that only messages generated by the communicating parties will be valid (except with negligible probability).

Let $\Pi_E = (\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ be a CPA-secure encryption scheme and $\Pi_M = (\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ a secure message authentication code. The construction is as follows:

CONSTRUCTION 4.17 CCA-secure encryption.

Define a CCA-secure encryption scheme as follows:

- $\mathsf{Gen}'(1^n)$: upon input 1^n, choose $k_1, k_2 \leftarrow \{0,1\}^n$
- $\mathsf{Enc}'_k(m)$: upon input key (k_1, k_2) and plaintext message m, compute $c = \mathsf{Enc}_{k_1}(m)$ and $t = \mathsf{Mac}_{k_2}(c)$ and output the pair (c, t)
- $\mathsf{Dec}'_k(c, t)$: upon input key (k_1, k_2) and ciphertext (c, t), first verify that $\mathsf{Vrfy}_{k_2}(c, t) = 1$. If yes, then output $\mathsf{Dec}_{k_1}(c)$; if no, then output $\perp$.

Before stating and proving the security of this construction, we introduce an additional requirement on the MAC scheme, called unique tags. Simply, a message authentication code has unique tags if for every k and every m, there is a *single value* t such that $\mathsf{Mac}_k(m) = t$. Note that the fixed-length MAC of Construction 4.3 has unique tags, whereas the variable-length MAC of Construction 4.5 does not. Nevertheless, this is not really a limitation because both the CBC-MAC and HMAC constructions have unique tags. We are now ready to state the theorem.

THEOREM 4.18 *Assume that* $\Pi_E = (\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ *is a CPA-secure encryption scheme and that* $\Pi_M = (\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ *is a secure message authentication code with unique tags. Then, Construction 4.17 is a CCA-secure encryption scheme.*

PROOF The idea behind the proof of this theorem is as follows. Since $(\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ is a secure message authentication code, we can assume that all queries to the decryption oracle are *invalid*, unless the queried ciphertext was previously obtained by the adversary from its encryption oracle. Therefore, the security of the scheme $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ in Construction 4.17 is reduced to the CPA-security of $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ (because the decryption oracle is effectively useless). In more detail, we first prove that except with negligible probability, the only valid queries made by the adversary to the decryption oracle are ciphertexts that were previously obtained from the encryption oracle. Then, given this claim, we prove that if the CCA-secure scheme is not secure, then neither is the underlying CPA-scheme $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$. This is due to the fact that an adversary for the CPA-secure scheme can actually simulate a decryption oracle for the CCA adversary. This simulation works by returning $\perp$ if the received ciphertext was never queried before, and returning the appropriate message if the ciphertext was generated by querying the encryption oracle. The validity of this simulation follows from the above claim. We now proceed to the formal proof.

Let $\mathcal{A}$ be any probabilistic polynomial-time CCA adversary attacking Construction 4.17. Define $\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)$ to be the event that in the experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$, the adversary $\mathcal{A}$ generates a query (c,t) to the decryption oracle that was not obtained from the encryption oracle and does not result in an oracle reply $\perp$. We claim that $\Pr[\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)]$ is at most negligible. Intuitively, this is due to the fact that if the oracle does not reply $\perp$, then t is a valid MAC tag for c. Thus, if (c,t) was not obtained by querying the encryption oracle, this means that $\mathcal{A}$ must have forged a MAC. Formally, we prove that if the probability that VALID-QUERY occurs is non-negligible, then we can construct an adversary $\mathcal{A}_{\text{mac}}$ that breaks the MAC as follows. Let $q(\cdot)$ be a polynomial that upper-bounds the running-time of $\mathcal{A}$ (and thus the number of oracle queries it makes). Then, adversary $\mathcal{A}_{\text{mac}}$, interacting in $\mathsf{Mac\text{-}forge}_{\mathcal{A}_{\text{mac}},\Pi_M}(n)$, chooses a random key k_1 for Enc and a random value

$i \leftarrow \{1, \ldots, q(n)\}$, and invokes the CCA-adversary $\mathcal{A}$. Adversary $\mathcal{A}_{\text{mac}}$ then simulates the encryption and decryption oracles for $\mathcal{A}$. The way it does this is to use k_1 and its MAC-generating oracle to simulate the encryption oracle for $\mathcal{A}$. Regarding the decryption oracle, all but the i^{th} query will be assumed to be invalid, and $\mathcal{A}_{\text{mac}}$ will "hope" that the i^{th} query is valid. In this case, $\mathcal{A}_{\text{mac}}$ will hope to have obtained a forged tag. More specifically, when $\mathcal{A}$ queries the encryption oracle with m, adversary $\mathcal{A}_{\text{mac}}$ computes $c = \mathsf{Enc}_{k_1}(m)$ and requests a tag t for c. Adversary $\mathcal{A}_{\text{mac}}$ then returns the pair (c, t) to $\mathcal{A}$ as its oracle reply. In contrast, in every decryption oracle query (c, t) from $\mathcal{A}$ *apart from the* i^{th} *one*, adversary $\mathcal{A}_{\text{mac}}$ first checks if (c, t) was ever generated from an encryption query. If yes, $\mathcal{A}_{\text{mac}}$ returns the plaintext m that was queried by $\mathcal{A}$ when (c, t) was generated. If not, $\mathcal{A}_{\text{mac}}$ returns $\perp$. In contrast, for the i^{th} decryption oracle query (c, t), adversary $\mathcal{A}_{\text{mac}}$ outputs (c, t) as its MAC forgery and halts. (We remark that the generation of the challenge ciphertext from the pair (m_0, m_1) is also carried out by $\mathcal{A}_{\text{mac}}$ for $\mathcal{A}$ as in the CCA experiment.)

Clearly $\mathcal{A}_{\text{mac}}$ runs in probabilistic polynomial-time. We now analyze the probability that $\mathcal{A}_{\text{mac}}$ generates a good forgery, and so succeeds in $\mathsf{Mac\text{-}forge}$. By our contradicting assumption, with non-negligible probability, adversary $\mathcal{A}$ generates a query (c, t) to the decryption oracle that was not obtained from the encryption oracle, and does not return $\perp$. We remark that since $(\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ has unique tags, it follows that the query c was never asked by $\mathcal{A}_{\text{mac}}$ to its MAC-tag oracle (because (c, t) was not obtained from an encryption query and there is only a single possible t that is a valid MAC tag for c). Therefore, such a pair (c, t) is a "good forgery" for $\mathcal{A}_{\text{mac}}$. Now, if all the decryption oracle queries generated by $\mathcal{A}$ up until the i^{th} one were indeed invalid, then the simulation by $\mathcal{A}_{\text{mac}}$ for $\mathcal{A}$ up until the i^{th} query is perfect. Furthermore, the probability that the i^{th} query is the *first* valid one generated by $\mathcal{A}$ is at least $1/q(n)$ because $\mathcal{A}$ makes at most $q(n)$ oracle queries, and one of these is the first valid one. Therefore, the probability that $\mathcal{A}_{\text{mac}}$ succeeds in $\mathsf{Mac\text{-}forge}$ is at least $1/q(n)$ times the probability that the VALID-QUERY event occurs. Since $\mathcal{A}_{\text{mac}}$ can succeed in $\mathsf{Mac\text{-}forge}$ with at most negligible probability, it follows that VALID-QUERY occurs with at most negligible probability. That is, we have that for some negligible function negl,

$$\Pr\left[\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)\right] < \mathsf{negl}(n).$$

Given that VALID-QUERY occurs with at most negligible probability, we now show that Construction 4.17 is CCA-secure. In this part of the proof, we reduce the security to the CPA-security of $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$. Specifically, let $\mathcal{A}$ be any probabilistic polynomial-time adversary for $\mathsf{PrivK}^{\mathsf{cca}}$. We use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{\text{enc}}$ for the CPA experiment with $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$. Adversary $\mathcal{A}_{\text{enc}}$ chooses a key k_2 and invokes the adversary $\mathcal{A}$. Whenever $\mathcal{A}$ asks an encryption query m, adversary $\mathcal{A}_{\text{enc}}$ queries its encryption oracle with m and receives back some c. Then $\mathcal{A}_{\text{enc}}$ computes $t = \mathsf{Mac}_{k_2}(c)$ and hands $\mathcal{A}$ the pair (c, t). Whenever $\mathcal{A}$ asks for a decryption query (c, t), $\mathcal{A}_{\text{enc}}$ checks if

(c,t) was generated in a previous encryption query. If yes, $\mathcal{A}_{\text{enc}}$ hands $\mathcal{A}$ the value m that was queried when (c,t) was generated. If no, $\mathcal{A}_{\text{enc}}$ hands $\mathcal{A}$ the response $\perp$. When $\mathcal{A}$ outputs a pair (m_0, m_1), adversary $\mathcal{A}_{\text{enc}}$ outputs the same pair and receives back a challenge ciphertext c. As above, $\mathcal{A}_{\text{enc}}$ hands $\mathcal{A}$ the challenge ciphertext (c,t) where $t = \mathsf{Mac}_{k_2}(c)$. Notice that $\mathcal{A}_{\text{enc}}$ does not need a decryption oracle because it assumes that any new query is always invalid. Furthermore, $\mathcal{A}_{\text{enc}}$ runs in probabilistic polynomial-time because it just invokes $\mathcal{A}$ and adds MAC tags (that are efficiently computable because $\mathcal{A}_{\text{enc}}$ chose k_2). It is straightforward to see that the success of $\mathcal{A}_{\text{enc}}$ in $\mathsf{PrivK}^{\mathsf{cpa}}$ when VALID-QUERY does not occur *equals* the success of $\mathcal{A}$ in $\mathsf{PrivK}^{\mathsf{cca}}$ when VALID-QUERY does not occur. That is,

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_{\text{enc}},\Pi_E}(n) = 1 \wedge \neg\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)] \\ = \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \neg\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)]]\end{aligned}$$

implying that

$$\begin{aligned}&\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_{\text{enc}},\Pi_E}(n) = 1] \\ &\geq \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_{\text{enc}},\Pi_E}(n) = 1 \wedge \neg\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)] \\ &= \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \neg\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)]\end{aligned} \tag{4.5}$$

Assume now by contradiction that there exists a *non-negligible function* ε such that

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

By the fact that $\Pr[\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)]$ is negligible, we have that it is smaller than $\varepsilon(n)/2$. This in turn implies that

$$\Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)] < \varepsilon(n)/2$$

and so

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1] &= \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \neg\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)] \\ &\quad + \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)] \\ &< \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \neg\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)] + \frac{\varepsilon(n)}{2}.\end{aligned}$$

Rearranging the above, and using the fact that $\mathcal{A}$ succeeds in $\mathsf{PrivK}^{\mathsf{cca}}$ with probability $1/2 + \varepsilon(n)$, we have that

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \neg\text{VALID-QUERY}_{\mathcal{A},\Pi'}(n)] &> \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1] - \frac{\varepsilon(n)}{2} \\ &= \frac{1}{2} + \frac{\varepsilon(n)}{2}.\end{aligned}$$

Combining this with Equation (4.5), we have that

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

implying that $\mathcal{A}_{\text{enc}}$ succeeds in $\mathsf{PrivK}^{\mathsf{cpa}}$ with non-negligible advantage over 1/2. Since this contradicts the CPA-security of $\Pi_E = (\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$, we conclude that Construction 4.17 is CCA-secure. ■

CCA-security and unique tags. If the MAC scheme does not have unique tags, then it may be easy to break the CCA scheme by simply modifying the MAC tag of the challenge ciphertext so that it is different to the given one, but is still valid. In such a case, it is possible to query the decryption oracle with the modified ciphertext, and the plaintext m_b will be returned. This seems rather artificial and actually demonstrates that full CCA-security may be an overkill, possibly motivating the search for meaningful relaxations of the notion. In any case, in the private-key encryption setting we have highly efficient constructions (i.e., Construction 4.17) and thus there is no reason to use relaxations.

CCA-security in real life. In some sense, Construction 4.17 is somewhat unsatisfying. It appears to bypass the meaning of CCA-security by preventing the adversary from ever using its decryption oracle. However, this is exactly the point! If in real life the adversary ever manages to achieve the effect of a decryption oracle, then it can only be used to obtain decryptions of ciphertexts sent by the legitimate parties. (We remark that this is unavoidable in any case. Furthermore, in real life, the decryption oracle that an adversary can obtains is typically limited and an adversary would usually not be able to learn anything by feeding captured ciphertexts. See the discussion on CCA-security in Section 3.7.)

4.9 * Obtaining Privacy and Message Authentication

In Chapter 3, we studied how it is possible to encrypt messages, thereby guaranteeing *privacy*. Until now in this chapter, we have showed how to generate secure message authentication codes, thereby guaranteeing *data authenticity* or *integrity*. However, sometimes we actually need both privacy and authentication. It may be tempting to think that if we use a secure encryption scheme and a secure MAC, then any combination of them should provide both privacy and authentication. However, this is unfortunately not at all the case. In general, even excellent cryptographic tools can be applied in a way so that the result is not secure. The unfortunate state of affairs is actually that it is very hard to combine cryptographic tools correctly. Thus, unless a specific combination has been proven secure, it is unwise to use it.

There are three common approaches to combining encryption and message authentication. We will consider each of the three. Let $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ be

an encryption scheme and let $(\mathsf{Gen}_E, \mathsf{Mac}, \mathsf{Vrfy})$ be a message authentication code. We will denote by k_1 an encryption key, and by k_2 a MAC key. The three approaches are:

1. *Encrypt-and-authenticate:* In this method, encryption and message authentication are computed and sent separately. That is, given a message m, the final message is the pair (c, t) where:

$$c = \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t = \mathsf{Mac}_{k_2}(m)$$

2. *Authenticate-then-encrypt:* Here a MAC tag t is first computed, and then the message and tag are encrypted together. That is, the message is c, where:

$$c = \mathsf{Enc}_{k_1}(m, t) \quad \text{and} \quad t = \mathsf{Mac}_{k_2}(m)$$

   Note that t is not sent separately to c, but is rather incorporated into the plaintext.

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 (c, t) where:

$$c = \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t = \mathsf{Mac}_{k_2}(c)$$

In this section we analyze each of these approaches. We stress that our analysis follows an *all or nothing* approach. That is, we require a scheme that will provide both privacy and authentication *for every possible secure encryption scheme and message authentication code*. Thus, our analysis will reject any combination for which there exists even a single counter-example. For example, we will show that "encrypt and authenticate" is not necessarily secure. This does *not* mean that for every encryption scheme and MAC, the combination is not secure. Rather it means that there *exists* an encryption scheme and MAC for which the combination is not secure. The reason that we insist that security should hold for all schemes is due to the fact that it should be possible to replace any secure encryption scheme with another one (and likewise MAC) without affecting the security of applications that use the scheme. We remark that such replacements are common in practice when cryptographic libraries are modified or updated, or standards are modified (as in the transition from 3DES to AES – see Section 5).

Encryption only versus encryption and authentication. Before proceeding, we briefly discuss the issue of when encryption alone is enough, and when it is necessary to both encrypt and authenticate. Clearly, when both privacy and integrity are needed, then combined encryption and authentication is essential. Thus, most online tasks, and clearly any online purchase or bank transaction, needs to be 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 to authenticate. However, it may be possible to inflict significant damage if financial reports and so on are modified (e.g., thereby causing a company to mistakenly publish false reports). We believe that it is best practice to *always encrypt and authenticate by default*; encryption-only should not be used unless you are absolutely sure that no damage can be caused by undetected modification of your files. There are also surprising cases where the lack of authentication can result in a breach of privacy, especially when it comes to network traffic.

The building blocks. The first question to be considered when discussing combinations of encryption schemes and message authentication codes, is what level of security is required from the encryption scheme and message authentication code that are used in the combination. We will consider the case that the encryption scheme is indistinguishable under chosen-plaintext attacks and the message authentication code is existentially unforgeable under chosen-message attacks. As will be shown below, this suffices for obtaining chosen-ciphertext security together with existential unforgeability.

Security requirements. In order to analyze which of the combinations of encryption and authentication are secure, we have to 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 refer to indistinguishability in the presence of chosen-ciphertext attacks and existential unforgeability in the presence of chosen-message attacks. Nevertheless, the definition and analysis suffice for understanding the key issues at hand.

Let $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme and let $(\mathsf{Gen}_E, \mathsf{Mac}, \mathsf{Vrfy})$ be a message authentication code. A combination of $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ and $(\mathsf{Gen}_E, \mathsf{Mac}, \mathsf{Vrfy})$ is a tuple of algorithms $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}', \mathsf{Vrfy}')$ as follows:

- *Key generation* Gen'*:* upon input 1^n, the key-generation algorithm G chooses $k_1, k_2 \leftarrow \{0,1\}^n$, and outputs (k_1, k_2).

- *The combination algorithm* EncMac'*:* the combination algorithm receives a pair of 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 $\mathsf{Mac}_{k_2}(\cdot)$.

- *Verification algorithm* Vrfy'*:* the verification algorithm receives a pair of keys (k_1, k_2) and a value c, and applies some combination of $\mathsf{Enc}_{k_1}(\cdot)$ and $\mathsf{Mac}_{k_2}(\cdot)$. At the end, Vrfy' outputs 1 or 0. (Notice that unlike in

the MAC setting, Vrfy' does not receive a plaintext message as well as a tag. Rather, all the necessary information is contained in c.)

- *Decryption algorithm* Dec'*:* the decryption algorithm receives a pair of keys (k_1, k_2) and a value c, and applies some combination of $\mathsf{Enc}_{k_1}(\cdot)$ and $\mathsf{Mac}_{k_2}(\cdot)$. At the end, Dec' outputs some value m.

The non-triviality requirement is that for every n and every pair of keys $k_1, k_2 \in \{0,1\}^n$, and for every value $m \in \{0,1\}^*$,

$$\mathsf{Dec}'_{k_1,k_2}(\mathsf{EncMac}'_{k_1,k_2}(m)) = m \quad \text{and} \quad \mathsf{Vrfy}'_{k_1,k_2}(\mathsf{EncMac}'_{k_1,k_2}(m)) = 1$$

As we have mentioned, our definition of security is simple. We require that the combination $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}', \mathsf{Vrfy}')$ is *both* a CCA-secure encryption scheme and a secure MAC. Notice that we actually "boost" security, because we begin only with a CPA-secure encryption scheme. We have the following definition:

DEFINITION 4.19 *We say that* $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}', \mathsf{Vrfy}')$ *is a* secure combination of encryption and authentication *if* $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}')$ *has indistinguishable encryptions under chosen-ciphertext attacks and the scheme* $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Vrfy}')$ *is existentially unforgeable under chosen message attacks.*

Definition 4.19 essentially says that a combination is secure if it does not harm the encryption security or the message authentication security. That is, each goal holds separately. We now analyze the three approaches for combining encryption and authentication mentioned above.

Encrypt-and-authenticate. As we have mentioned, in this approach an encryption and message authentication code are computed and sent separately. That is, given a message m, the final message is the pair (c, t) where

$$c = \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t = \mathsf{Mac}_{k_2}(m)$$

This combination does *not* yield a secure encryption scheme. In order to see this, first notice that at least by definition, a secure MAC does not necessarily imply privacy. Specifically, if $(\mathsf{Gen}_E, \mathsf{Mac}, \mathsf{Vrfy})$ is a secure message authentication code, then so is the scheme defined by $\mathsf{Mac}'_k(m) = (m, \mathsf{Mac}_k(m))$. The important point is that Mac' reveals the message m completely. Therefore, for *any encryption scheme*, the combination $(\mathsf{Enc}_{k_1}(m), \mathsf{Mac}'_{k_2}(m))$ completely reveals m and is therefore not indistinguishable. Note that this is true even if $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ is itself CCA-secure, rather than only being CPA-secure.

This example is enough because by our requirements, a secure combination must be secure for *any* instantiation of the underlying building blocks. Nevertheless, a counter-example for this combination exists for message authentication codes used in practice. In Exercise 4.15 you are asked to demonstrate this fact.

Authenticate-then-encrypt. In this approach, a MAC tag $t = \mathsf{Mac}_{k_2}(m)$ is first computed, and then the pair (m, t) is encrypted. Finally, the ciphertext $\mathsf{Enc}_{k_1}(m, \mathsf{Mac}_{k_2}(m))$ is sent. We will now show that this combination is also *not* necessarily secure. The counter-example here is somewhat contrived but as we will discuss below, it suffices to show that the method should not be used. The counter-example uses 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.[4] The decoding of a message works by mapping 00 back to 0, and 01 and 10 back to 1. However, a pair of bits 11 will result in decoding the message back to $\perp$ (irrespective of the other bits).

- Define $\mathsf{Enc}'_k(m) = \mathsf{Enc}_k(\mathsf{Transform}(m))$, where Enc is a stream cipher that works by generating a new pseudorandom stream for each message to encrypt, and then XORs the stream with the input message. For example, this can be obtained by using a pseudorandom function (secure block cipher) in CTR mode, with a new random IV for each encryption. Note that both the encryption schemes Enc and Enc' are CPA secure.

We now show that the combination of the above encryption scheme with any MAC is not secure in the presence of chosen-ciphertext attacks. In fact, the attack that we will show works as long as an adversary can find out if a given ciphertext is valid, even if it cannot obtain full decryptions. (As we have discussed in Section 4.8, such an attack is very realistic.)

Let $\mathcal{A}$ be an adversary for the CCA-experiment that works as follows. 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 c (i.e., takes their complement), and verifies if the resulting ciphertext is valid. (Technically, in the CCA setting this query can be made to the decryption oracle. However, the adversary only needs to know if the ciphertext is valid or not, and so a weaker oracle would also suffice.) If the ciphertext is valid, then the adversary knows that the first bit of the message sent equals 1. This is due to the fact that if the first bit of m equals 1, then the first two bits of $\mathsf{Transform}(m)$ can be 01 or 10. Thus, the complement of these two bits still maps to the same initial bit 1. In contrast if the ciphertext is not valid, then the adversary knows that the first bit of m equals 0. This is due to the fact that 0 is mapped to 00 and so flipping these bits results in 11, which is an incorrect encoding. Thus, the plaintext is $\perp \neq m$ but the MAC is still computed over m.

We remark that flipping bits in the ciphertext results in exactly the same effect in the plaintext due to the fact that Enc' is based on a stream cipher.

[4] We remark that this encoding is contrived. However, encodings of initial inputs are often used and we would not like the security of a cryptographic scheme to depend on which encoding is used, if any.

We also note that this attack can be carried out on each bit separately, if desired, with the result being a full decryption of the ciphertext.

We stress that this counter-example demonstrates that the authenticate-then-encrypt combination is *not always* secure. However, there are some instantiations that are secure (for example, the specific encryption scheme and MAC used within SSL are secure); see [86]. Nevertheless, as mentioned above, it is bad practice to use a methodology whose security depends on specific implementations.

Encrypt-then-authenticate. In this approach, an encryption is first computed, and then the MAC is computed over the ciphertext. That is, the message is the pair (c, t) where

$$c = \mathsf{Enc}_{k_1}(m) \quad \text{and} \quad t = \mathsf{Mac}_{k_2}(c)$$

We have the following theorem:

THEOREM 4.20 *Let* $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ *be any encryption scheme that is secure under a chosen plaintext attack, and let* $(\mathsf{Gen}_M, \mathsf{Mac}, \mathsf{Vrfy})$ *be any message authentication code that is existentially unforgeable under chosen-message attacks. Then,* $(\mathsf{Gen}', \mathsf{EncMac}', \mathsf{Dec}', \mathsf{Vrfy}')$ *derived by the encrypt-then-authenticate methodology with these schemes is a secure combination of encryption and authentication.*

Note that $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ is only CPA-secure, but the combination is CCA-secure. This should not come as a surprise given Construction 4.17 that follows the encrypt-then-authenticate methodology and is CCA-secure. We do not present the proof of this theorem and leave it as an exercise. The fact that the combination is CCA-secure has already been proven in Theorem 4.18. The fact that it is a secure MAC can be proven by a rather straightforward reduction to the MAC scheme (in a similar way to the first half of the proof of Theorem 4.18).

Secure combinations versus CCA-security. We note that although we use the same construction for achieving CCA-security and combining privacy and encryption, the security goals in both cases are different. Namely, in the setting of CCA-security, we are not necessarily interested in obtaining authentication. Rather, we wish to ensure privacy even in a strong adversarial setting where the adversary is able to obtain some information about a plaintext from a given ciphertext. In contrast, when considering secure combinations, we are interested in both goals of CCA-security and authenticity. Clearly, as we have defined it, a secure combination provides CCA-security. However, the opposite direction is not necessarily true.

Independent keys. We conclude by stressing a basic principle of security

and cryptography: *different security goals should always use different keys.*[5] That is, if an encryption scheme and a message authentication scheme are both needed, then independent keys should be used for each one. In order to illustrate this here, consider what happens to the *encrypt-then-authenticate* methodology when the same key k is used both for the encryption scheme $(\mathsf{Gen}_E, \mathsf{Enc}, \mathsf{Dec})$ and for the message-authentication code $(\mathsf{Gen}_E, \mathsf{Mac}, \mathsf{Vrfy})$. We provide a concrete example using a pseudorandom-permutation based encryption scheme and a pseudorandom-permutation based MAC. That is, let F be a pseudorandom permutation. Then, it follows that both F and F^{-1} are pseudorandom permutations. Define $\mathsf{Enc}_k(m) = F_k(r\|m)$ for $m \in \{0,1\}^{n/2}$ and a random $r \leftarrow \{0,1\}^{n/2}$, and define $\mathsf{Mac}_k(c) = F_k^{-1}(c)$. (This encryption scheme is different from the one we defined in Construction 3.25. Nevertheless, using a similar analysis it can be shown to be CPA-secure. In fact, it is even CCA-secure.) Clearly, Enc is CPA-secure and Mac is a secure MAC. However, the combined encryption and authentication of a message m with the same key k yields:

$$\mathsf{Enc}_k(m), \mathsf{Mac}_k(\mathsf{Enc}_k(m)) = F_k(m\|r), F_k^{-1}(F_k(m\|r)) = F_k(m\|r), (m\|r)$$

Thus, the message m is revealed in the output.

References and Additional Reading

The definition of security for message authentication codes was adapted by Bellare et al. [17] from the definition of security for digital signatures given by Goldwasser et al. [71] (see Chapter 12). The basic paradigm of using pseudorandom functions for message authentication was introduced by Goldreich et al. [68], and Construction 4.5 for extending a fixed-length MAC to a variable-length MAC was shown by Goldreich in [66]. An alternative of 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. CBC-MAC was standardized in the early '80s [133, 9] and was later formally analyzed and proven secure by Bellare et al. [17] 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. [14] and later became a standard [104].

Collision-resistant hash functions were formally defined by Damgård [46]. The Merkle-Damgård transform was introduced independently by Damgård and Merkle [47, 96]. For further information about SHA-1 and MD5, see, e.g.,

[5] We note that it is sometimes possible to use the same key for different goals; however, an explicit proof is needed for such cases.

the textbook by Kaufman, et al. [84]. Note, however, that their treatment pre-dates the recent attacks by Wang et al. [129, 128]. For other interesting applications of collision-resistant hash functions in computer security, see Kaufman et al. [84] (but beware that in many cases the security arguments are heuristic only). 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".[6]

The notion of chosen-ciphertext attacks was first formally considered by Naor and Yung [99] and then by Rackoff and Simon [109]. The method of encrypting and then applying a MAC for achieving CCA-security was described by Dolev et al. [52]. Analyses of the different ways of applying encryption and message authentication for simultaneously achieving privacy and authentication were given by [18] and Krawczyk [86].

Exercises

4.1 Consider the following fixed-length MAC scheme with length parameter $\ell(n) = 2n - 2$ that uses a pseudorandom function F. Algorithm Gen chooses $k \leftarrow \{0,1\}^n$. Upon input $m \in \{0,1\}^{2n-2}$, 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.

4.2 Consider a "CCA-type" extension of the definition of secure message authentication codes where the adversary is provided with both a Mac and Vrfy oracle.

(a) Provide a formal definition and explain why such a notion may make sense.

(b) Show that when the Mac scheme is deterministic, your definition is equivalent to Definition 4.2.

(c) Show that when the Mac scheme may be probabilistic, the definitions are not equivalent. (That is, show that there exists a probabilistic scheme that is secure by Definition 4.2 but not by your definition.)

4.3 Prove that Construction 4.5 remains secure for each of the following modifications:

[6] http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html.

(a) Instead of using a pseudorandom function, use any fixed-length MAC with the appropriate parameters.

(b) Instead of including d 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?

(c) Instead of outputting $(r, t_1, \ldots, t_d)$, output $(r, \oplus_{i=1}^{d} t_i)$. What is the advantage of this modification? (This is a more challenging modification to prove.)

4.4 Prove that the basic CBC-MAC construction is *not* secure if a random IV is used or if all the blocks are output (rather than one). Demonstrate this even for the case of fixed-length messages. Try and find a real-life scenario where the attack on the case of a random IV can be utilized.

4.5 Show that the basic CBC-MAC construction is *not* secure when considering messages of different lengths.

4.6 Provide formal definitions for second preimage resistance and preimage resistance. Then formally prove that any hash function that is collision resistant is second preimage resistant, and any hash function that is second preimage resistant is preimage resistant.

4.7 Generalize the Merkle-Damgård construction so that it can work for any function that compresses by at least one bit. You should refer to a general input length ℓ' and general output length ℓ.

4.8 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 obtaining s_1 and s_2, respectively. Then, $H^{s_1,s_2}(x) = H^{s_1}(x)\|H^{s_2}(x)$.

(a) Prove that if at least one of (Gen_1, H_1) and (Gen_2, H_2) are collision resistant, then (Gen, H) is collision resistant. Why is such a construction useful?

(b) Show that an analogous claim cannot be made regarding second preimage and preimage resistance.

Hint: You may use contrived functions to demonstrate this.

4.9 Let (Gen, H) be a collision-resistant hash function. Is the function $(\mathsf{Gen}, \hat{H})$ defined by $\hat{H}^s(x) = H^s(H^s(x))$ necessarily collision resistant?

4.10 Provide a formal proof of Theorem 4.12 (i.e., describe and prove the formal reduction).

4.11 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 $h^s(z_B \| L)$).

(b) Modify the construction so that instead of outputting $z = h^s(z_B \| L)$, the algorithm outputs $z = (z_B \| L)$.

(c) Instead of using a fixed IV, choose $IV \leftarrow \{0,1\}^n$ and define $z_0 = IV$. Then, set the output to be $z = (IV, h^s(z_B \| L))$.

(d) Instead of using a fixed IV, just start the computation from x_1. That is, define $z_1 = x_1$ and compute $z_i = h^s(z_{i-1} \| x_i)$ for $i = 2, \ldots, B$.

4.12 Provide a full and detailed specification of HMAC for arbitrary length inputs and outputs ℓ' and ℓ of the underlying compression function. Describe the instantiation of HMAC with SHA-1.

4.13 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.

4.14 Show that Construction 4.17 is CCA-secure even when the MAC of Construction 4.5 is used (recall that this MAC does not have unique tags). Try and come up with a general requirement for the MAC used that includes those with unique tags and Construction 4.5.

> **Hint:** The standard notion of a MAC forgery is that as m must not have been queried to the oracle. Thus, if $\mathcal{A}$ queries m and receives back t, then it is not considered a forgery if it outputs (m, t') where $t' \neq t$ and $\mathsf{Vrfy}_k(m, t) = 1$. Consider a stronger notion of forgery that state that $\mathcal{A}$ must not even be able to output (m, t') as above. That is, even after seeing one valid MAC tag t for a message m it should be hard to generate another valid MAC tag for m.

4.15 Show that for both the CBC-MAC and HMAC constructions, the encrypt-and-authenticate method is not secure. In particular, show that it is not CPA-secure or even indistinguishable for multiple messages and eavesdropping adversaries. Describe where this attack can be used in real life. (You are not allowed to use encodings as for the authenticate-then-encrypt method; there is a much simpler attack here.)

4.16 Prove Theorem 4.20.

> **Hint:** The fact that the combination is CCA-secure has already been proven in Theorem 4.18. The fact that it is a secure MAC can be proven by a rather straightforward reduction to the MAC scheme, in a similar way to the first half of the proof of Theorem 4.18.

Chapter 5

Pseudorandom Objects in Practice: Block Ciphers

In previous chapters, we have studied how pseudorandom permutations (or block ciphers, as they are often called) 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. In fact, an even more basic questions to ask is whether they even exist. We stress that the mere existence of pseudorandom functions and permutations (under widely-believed assumptions) is within itself amazing. In Chapter 6 we study this question from a theoretical point of view and show under what (minimal) assumptions it is possible to construct pseudorandom permutations. In contrast, in this chapter, we show how block ciphers (i.e., pseudorandom permutations) are constructed in practice. We stress that unlike previously studied material, there will be no formal proof or justification about why the constructions in this chapter are secure. The reason for this is that the constructions presented here are in fact *heuristic* and have no proof of security. Furthermore, there is no known reduction of the security of these constructions to some other problem. Nevertheless, a number of the block ciphers that are used in practice have withstood many years of public scrutiny and attempted cryptanalysis. Given this fact, it is reasonable to just assume that these tried and tested block ciphers are indeed pseudorandom permutations, and thus can be used in our proven constructions of secure encryption and message authentication codes.

One may wonder why it makes sense to rigorously prove the security of an encryption scheme based on pseudorandom permutations, when in the end the pseudorandom permutation is instantiated with a completely heuristic construction. There are actually a number of reasons why this methodology makes sense. We outline three of these reasons:

- A pseudorandom permutation is a relatively low-level primitive, meaning that its security requirements are simple to understand and relatively simple to test.[1] This makes cryptanalysis of a candidate pseudorandom

[1] We remark that our categorization of a pseudorandom permutation as "low-level" is a *relative* one. Relative to the assumptions used in Chapter 6, pseudorandom permutations are

permutation easier, thus raising our confidence in its security if it is not broken. (Compare the task of testing if a function behaves pseudorandomly to the task of testing if an encryption scheme provides security in the presence of chosen-ciphertext attacks. We argue that the latter is often more complex to analyze.)

- If a specific construction of a pseudorandom permutation is used and later broken, it can be more easily replaced than if the high-level application is broken. That is, if an encryption scheme uses a pseudorandom permutation with a specific length (say 128 bits), then the permutation can be replaced with any other construction of the same length without any change to the input/output format of the higher-level encryption scheme.

- Pseudorandom permutations have many applications (encryption, message authentication and others). By proving the security of higher level constructions that use pseudorandom permutations, we limit the amount of heuristics used to a single realm. This also means that cryptographers who design practical schemes can focus on constructing good pseudorandom permutations, without thinking about the specific application they will be used for.

We will now proceed to show how pseudorandom permutations are constructed in practice.

Block ciphers as encryption schemes or pseudorandom permutations. In many textbooks and other sources on cryptography, block ciphers are presented as encryption schemes. In contrast, we stress here that despite their name, block ciphers should be viewed as pseudorandom permutations and *not* as encryption schemes. Stated differently, we strongly advocate viewing block ciphers as basic building blocks for encryption (and other) schemes, and not as encryption schemes themselves. From here on in this chapter we will use the name "block cipher" in conformance with their popular name. We hope that this will not cause confusion regarding what they really are.

We remark that this view is very widespread today and modern block ciphers are constructed with the aim of being pseudorandom permutations. In order to see this, we diverge for a moment to the Advanced Encryption Standard (AES). The AES is a block cipher that was adopted in 2001 as a standard by the National Institute of Standards and Technology in the USA. Due to its importance, we will study it later in this chapter. At this point, we will just note that it was chosen after a lengthy competition in which many algorithms were submitted and analyzed. The call for candidate algorithms for the AES stated the following under the section on "evaluation criteria":

very high level. However, relative to encryption schemes via modes of operation and message authentication codes, we consider the building block of a pseudorandom permutation to be rather low level.

> *The security provided by an algorithm is the most important factor in the evaluation. Algorithms will be judged on the following factors:*
>
> 1. *Actual security of the algorithm compared to other submitted algorithms (at the same key and block size).*
> 2. *The extent to which the algorithm output is indistinguishable from a random permutation on the input block.*
> 3. *Soundness of the mathematical basis for the algorithm's security.*
> 4. *Other security factors raised by the public during the evaluation process, including any attacks which demonstrate that the actual security of the algorithm is less than the strength claimed by the submitter.*

Notice that the second factor explicitly states that the AES must be a *pseudorandom permutation*. Thus, as we have stated, modern block ciphers are intended to be pseudorandom permutations. As such, they are suited for use in all of the constructions that use pseudorandom permutations and functions that we have seen so far in this book.

Block ciphers in practice and Definition 3.28. Although we are going to consider block ciphers as pseudorandom permutations, practical constructions of block ciphers do not quite meet the definition. Specifically, practical block ciphers are typically only defined for one (or a few) different key and block lengths. This is in contrast to Definition 3.28 that refers to all possible key and block lengths. Nevertheless, we believe that the appropriate way to deal with this discrepancy is to consider only the lengths specified by the block cipher, and then to ensure that these lengths (or more exactly the values n taken) are large enough to maintain pseudorandomness in the presence of modern adversaries using modern powerful computers (and a large amount of computing time).

The aim and outline of this chapter. We stress that the aim of this chapter is *not* 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. There are excellent standardized block ciphers that are not patented, are highly efficient, and have undergone intensive scrutiny regarding their security (this is true of both DES and AES that we describe later in the chapter). Rather, the aim of this section is to provide a feeling or intuition as to how modern block ciphers are constructed and why. We will look at two different types of high-level constructions: *substitution-permutation networks* (first introduced by Shannon) and *Feistel structures* (introduced by Feistel). Following this, we will describe the DES and AES block ciphers, but will not provide full detailed descriptions. In particular, the descriptions that we provide here are not fully accurate and are definitely

not specifications that can be used for implementation. Given that our aim is to provide intuition as to how modern block ciphers are constructed, we don't feel that much is gained by providing full detailed specifications.

5.1 Substitution-Permutation Networks

As we have mentioned, the main property of a block cipher is that it should behave like a random permutation. Of course, a truly random permutation would be perfect. However, for a block cipher with input and output length of n bits, the size of the table needed for holding the random permutation is $n \cdot 2^n$ (actually, this is the size of the table needed for a random function, but it is not much smaller for a random permutation). 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 functions. The basic idea is to break the input up into small parts and then feed these parts through different small random functions. The outputs of these random functions are then mixed together, and the process is repeated again (for a given number of times). Each application of random functions followed by mixing is called a *round* of the network (the construction is often called a "network" and we keep this name for consistency with other sources). Note that by using small random functions, the lack of structure that is inherent to a random function is introduced into the construction, but without the heavy expense.[2] This paradigm introduced by Shannon is often called the *confusion-diffusion* approach (in fact, Shannon himself coined these terms in his original paper). Small random functions introduce "confusion" into the construction. However, in order to spread the confusion throughout, the results are mixed together, achieving "diffusion".

In order to see why diffusion is necessary, consider a block-cipher F_k that works by simply applying small random functions to each 8-bit portion of the input. Now, let x and x' be values that differ only in the first bit. It follows that $F_k(x)$ and $F_k(x')$ differ only in the first byte (because in all other bytes x equals x'). In a truly random function $F_k(x)$ and $F_k(x')$ should look completely different, because each output value in a random function is chosen uniformly and independently of all other output values. However, in

[2] Sixteen random functions with input/output of 8 bits each can be stored in only $16 \cdot 8 \cdot 2^8$ bits, or 32 Kbits. This is in contrast with a random functions with input/output of 128 bits, that requires $128 \cdot 2^{128}$ bits to store. We note that this latter number has 40 digits, whereas it is estimated that the number of atoms in Earth is a number with 49 digits. Thus, we could never even build a disk large enough to store such a table.

this imaginary block cipher where diffusion is not used, they are the same except for in the first byte. Thus, such a block cipher is very far from a pseudorandom function. The repeated use of confusion and diffusion ensures that any small changes in the input will be mixed throughout and so the outputs of similar inputs will look completely different.

Substitution-permutation networks. A substitution-permutation network is a direct implementation of this paradigm. The "substitution" portion refers to small random functions (much like the mono-alphabetic substitution cipher studied in Section 1.3 that is actually a random 1–1 and onto function of the alphabet), and "permutation" refers to the mixing of the outputs of the random functions. In this context, "permutation" refers to a reordering of the output bits (not to a 1–1 and onto function as we typically use it in this book). The small substitution functions are called *S-boxes* and we call the permutations that follow them *mixing permutations*. One question that you may be asking yourselves is where does the secret key come in? There are a number of possible answers to this, and different substitution-permutation networks can use different methods. One possibility is to have the key specify the S-boxes and mixing permutations. Another possibility is to mix the key into the computation in between each round of substitution-permutation (recall that the substitution-permutation operation is repeated many times). In this latter case, the S-boxes and mixing permutations are publicly known (in accordance with Kerckhoffs' principle; see Section 1.3). Our presentation here is in accordance with the latter approach, where the key is mixed into the intermediate results between each round. This is often just by a simple XOR operation. That is, the key is XORed with the intermediate result after each round of the network computation. We remark that typically it is not the same key that is XORed each time. Rather, different keys are used. To be more exact, the key to the block cipher is referred to as a *master key*, and subkeys for each round are derived from it; this derivation procedure is known as the *key schedule*. We remark that the key schedule is often very simple and may work by just taking subsets of the bits, although complex schedules can also be defined. See Figure **??** for the high-level structure of a substitution-permutation network, and Figure **??** for a closer look at a single round of such a network.

An important point to notice is that we have not specified at all how the S-boxes and mixing permutations should be chosen. We have also not specified whether the S-boxes and mixing permutations are the same or different in each round. We will *not* present design principles for the construction of "good" S-boxes (with one exception below), and likewise will not describe how the subkeys should be derived from the master key and how the mixing permutations should be chosen. We do stress that these choices are what determines whether the block cipher is trivially breakable or very secure. Nevertheless, as we have mentioned, our aim in this chapter is not to teach how to construct block ciphers. The contrary is true: we strongly believe that non-

experts (meaning those who do not have many many many years of experience in this) should never attempt such a task.

Despite the above there are two design choices that we will mention. The first is necessary for making the block cipher a permutation (i.e., a 1–1 and onto function), and the other is a basic requirement for security. Both choices relate to the S-boxes.

Design choice 1 – invertibility of the S-boxes. In a substitution-permutation network, the S-boxes must be invertible. In other words, they must be 1–1 and onto functions. The reason for this is that otherwise the block cipher will not be a permutation (i.e., it will not be 1–1 and onto). In order to see that making the S-boxes 1–1 and onto suffices, we show that given this assumption it is possible to fully determine the input given any output and the key. Specifically, we show that every round can be uniquely inverted (yielding that the entire network can be inverted by working from the end back to the beginning). For the sake of this description, we define a round to consist of three stages: XORing of the subkey with the input to the round, passing the input through the S-boxes, and mixing the result via the mixing permutation. The mixing permutation can easily be inverted because the permutation determines for every bit i in the input, the bit j where it appears in the output (thus the j^{th} bit is just reversed to the i^{th} bit). Given the output of the permutation, we therefore obtain the input to the permutation, which is exactly the output of the S-boxes. Given the fact that the S-boxes are 1–1 and onto functions, these too can be inverted by looking at the table that defines them. We therefore obtain the input to the S-boxes. Finally, the subkey can be XORed with this input and we obtain the value from the beginning of the round (note that the required subkey can be derived from the master key in the same way when inverting the block cipher as when computing it). We therefore have the following claim:

CLAIM 5.1 *In a substitution-permutation network F in which the S-boxes are all 1–1 and onto (and of polynomial-size), there exists an efficient procedure for computing $F^{-1}(y)$. Furthermore, for every key k and every input x, $F_k^{-1}(F_k(x)) = x$.*

In addition to the requirement that the S-boxes be uniquely invertible, it is clear that they must be random looking. One might therefore think that the best strategy is to simply choose them completely randomly, under the constraint that they be 1–1 and onto. However, this is actually not the best strategy, because truly random S-boxes do not provide the strongest protection against advanced cryptanalytic techniques (like differential and linear cryptanalysis that we briefly discuss below in Section 5.6).

Design choice 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 similar inputs will not look independent (whereas in a random permutation, the outputs of similar inputs are independently distributed). In order to ensure that this is the case, block ciphers are designed to have an *avalanche effect*, meaning that small changes in the input propagate quickly to very large changes in the intermediate values (and thus outputs). It is easy to demonstrate that the avalanche effect holds in a substitution-permutation network, provided that the following two properties hold:

1. *The S-boxes are designed so that any change of at least a single bit to the input to an S-box results in a change of at least two bits in the output.*
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.*

We remark that the first property is not necessarily obtained by small random functions or permutations. For example, consider the case that the S-box has input/output length of 4 bits (this may seem very small but the size of a table for an 8-bit S-box is often too large). Now, for any 4-bit input, there are 5 values that differ from the input by 1 or less bits (the input itself plus four other values obtained by flipping a single bit of the input). In contrast, there are 16 possible output values. Thus, the probability that a random output differs from the input by 1 or less bits is $\frac{5}{16} > \frac{1}{4}$. Given that a number of S-boxes may be used (e.g., DES has 8 of them), randomly chosen S-boxes are not likely to have this property.

Returning to the avalanche effect, let us assume that all the S-boxes and mixing permutations are chosen with the aforementioned properties. Consider now what happens when the block cipher is applied to two inputs that differ by only a single bit. For the sake of concreteness, assume that the S-boxes have input/output size of 4 bits, and that the block size is 128 bits. We track the computation of two similar inputs round by round:

1. After the first round of the network, the intermediate values differ in two places. This is due to the fact that the two inputs differ in only a single bit and so all the input values are the same except for in one S-box. Given the above S-box property, it follows that the outputs of this S-box differ in two places.

2. By the second property, the permutation at the end of the first round spreads the two different bits into different regions of the intermediate string. Therefore, it follows that at the beginning of the second round, there are two S-boxes that receive inputs that differ by one bit. Following the same argument as previously, we have that at the end of the second round, the intermediate values differ in 4 bits.

3. Continuing with the same argument, we have that the intermediate values differ by 8 bits after the 3rd round, 16 bits after the 4th round, 32 bits after the 5th round, 64 bits after the 6th round and 128 bits (i.e., everywhere) after the 7th round. Of course, we don't really mean that the bits are all different, but rather that they have all been affected and so no similarity remains.

We conclude that after i rounds, 2^i bits have been affected (thus for a 128 bit block, 7 rounds are required to complete the avalanche effect).

Pseudorandomness of substitution-permutation networks. As we have discussed, there is no formal justification for why such a design yields a pseudorandom permutation. Nevertheless, experience of many years shows that the confusion-diffusion paradigm works, 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 below in Section 5.5, has a similar structure to the substitution-permutation network described above, and is widely believed to be a very strong pseudorandom permutation.

Attacks on reduced-round substitution-permutation networks. 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 worthwhile seeing. They also show why a number of rounds are needed. Recall that according to Definition 3.28 (pseudorandom permutations), 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 guess what function is computed by its oracle. 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.

1. *Attack on a single-round substitution-permutation network:* We demonstrate this attack in a weaker adversarial model than above. Namely, we assume only that the adversary is given one input/output pair (and not that it can choose the inputs upon which the block cipher is computed). Let x be the input and y the output. We will demonstrate how the adversary can easily learn the secret key k for which $y = F_k(x)$, where F denotes the single-round substitution-permutation network. The adversary begins by inverting the mixing permutation, and then the S-boxes. It can do this because the specification of the permutation and S-boxes is public. The intermediate value that the adversary receives from these inversions is exactly $x \oplus k$ (by the design of a single substitution-permutation round). Since the adversary also has the input x, it immediately derives the secret key k. This is therefore a trivial complete break.

2. *Attack on a two-round substitution-permutation network:* In this case, we also show a complete break. In order to demonstrate the attack, we consider concrete parameters. Let the block size be 64 bits and let each S-box have input/output of size 4 bits (as we have mentioned, 8 is usually too large). Furthermore, let the key k be of length 128 bits where the first half of the key is used in the first round and the second half in the second round (let k^a and k^b denote these two 64 bit parts of the key). We use independent keys here to simplify the description of the attack below, but this only makes the attack "harder".

   Now, let x be the input and y the output (each of 64 bits). Denote $z = z_1, \ldots, z_{16}$, where each z_i is of length 4 bits (we will use this notation for x, y, k^a and k^b). The adversary begins by peeling off the last round, as in the attack on the single-round block cipher. Denote by w the value that it receives after inverting the mixing permutation and S-boxes of the second round. Denote $\alpha = w_1 \oplus k_1^b$ (of course, the adversary does not know k^b but it wishes to learn it). The important observation here is that when working from the input to the output, the value of α is influenced by at most 4 different S-boxes (because in the worst case, each bit of input 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 it. Next, notice that at most 16 bits of the key k^a influence the computation of these four S-boxes. It follows that the adversary can guess 16 bits of k^a and the four-bit portion k_1^b of the key k^b, and then *verify* the guess with the input-output (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, and then computing the appropriate 4 first-round S-boxes and 4 bits of the first-round mixing permutation. The value α obtained is then compared with $w_1 \oplus k_1^b$ (where k_1^b is also part of the guess). If equality is not obtained, then this guess of 16 bits of k^a and k_1^b is certainly incorrect. If equality is obtained, then this guess *may* be correct. However, it may also be incorrect (and equality is obtained by chance). Nevertheless, it is possible to use a number of input/output pairs and verify the guess of the key portion with all of the pairs.[3] For the sake of concreteness, assume that 8 pairs are used. It follows that 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 and we obtain a total complexity of $16 \cdot 2^{23} = 2^{27}$. We remark that in this process all of k^a is also learned and so the entire 128-bit key

[3] Since there are 2^{20} possible guesses of the key portion, and verification takes place with just 4 bits, we obtain that 2^{16} different keys are expected to pass the test with a single input/output pair. Assuming random behavior of the block cipher, we obtain that with 5 different (preferably random) input/output pairs, the verification takes place with 20 bits and so just a single key is expected to pass the test. Given 10 different input/output pairs, it is unlikely that the test will leave more than one possibility for the key portion.

$k = (k^a, k^b)$ is learned in only time 2^{27}. (In fact, the complexity will be even less because all of k^a will be learned after only 4 or so repetitions of the above procedure, and only k^b will remain unknown.)

We recommend sketching two rounds of a substitution-permutation network and tracing the steps of the above attack.

There is an important lesson to be learned from the above attack. Observe that the attack is made possible since different parts of the key can be isolated from the other parts (it is much quicker to carry out 16 attacks of time 2^{20} than a single attack of time 2^{128} or even 2^{64}). Thus, the *diffusion* step in the construction is also needed to make sure that all of the bits of the key affect all of the bits of the output. Two rounds of the network are not enough for this to happen.

3. *Attack on a three-round substitution-permutation network:* We present a weaker attack here; instead of attempting to learn the key, we just show that it is easy to distinguish a three-round block cipher from a pseudorandom permutation. This attack is based on the observation that the avalanche effect is not complete after only three rounds (of course, this depends on the block size and S-box size, 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 one bit. A three-round block cipher will have the property that many bits of the output of both inputs will be the same. Thus, it is clearly not a pseudorandom function.

5.2 Feistel Networks

A Feistel network is an alternative way of constructing a block cipher. The low-level building blocks (S-boxes, mixing permutations and key schedule) are the same; the difference is in the high-level design. The advantage of Feistel networks over substitution permutation networks is that they enable the use of S-boxes that are not necessarily invertible. This is important because a good block cipher has chaotic behavior (and as such it looks random). However, requiring that all of the components of the construction be invertible inherently introduces structure, which contradicts the need for chaos. *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, how can you invert the overall structure). Nevertheless, the Feistel design ingeniously overcomes this obstacle.

A Feistel network refers to an internal f-function that does not need to be invertible. This function receives a subkey and typically contains components

like S-boxes and mixing permutations. In any case, the framework of a Feistel network can deal with *any* internal f-function, irrespective of its design. The input x to a Feistel network is separated into two halves, x_1 and x_2, and each half is passed separately through the f-function. Thus, for an n-block cipher, the f-function has input/output length of $n/2$. (We stress again that although the input and output lengths are the same, the function is not necessarily invertible and so, in particular, is not necessarily 1–1 and onto.) The mathematical definition of a Feistel network is given as follows:

1. For input x, denote by x_1 and x_2 the first and second halves of x, respectively.
2. Let $v_1 = x_1$ and $v_2 = x_2$.
3. For $i = 1$ to r (where r is the number of rounds in the network):
   (a) Let $w_1 = v_2$ and $w_2 = v_1 \oplus f_i(v_2)$, where f_i denotes the f-function in the i^{th} round of the network.
   (b) Let $v_1 = w_1$ and $v_2 = w_2$.
4. The output y is (v_1, v_2).

See Figure **??** for a 4-round Feistel network (it is easier to understand how it works by looking at the diagram, and then only afterwards at the mathematical definition above).

Inverting a Feistel network. Recall that the f-function is not necessarily invertible and may not even be 1–1. Thus in order to invert the block cipher, we cannot rely on the ability to compute f^{-1}. Rather, the block cipher can be inverted while computing the f-function in a forward manner only. In order to see this, let (α_i, β_i) be the intermediate values at the beginning of round i and let $(\alpha_{i+1}, \beta_{i+1})$ be the intermediate values at the beginning of round $i + 1$. Then, it holds that:

$$\alpha_i = \beta_{i+1} \oplus f_i(\alpha_{i+1}) \qquad \text{and} \qquad \beta_i = \alpha_{i+1} \tag{5.1}$$

This can easily be seen by following the network in Figure **??** (note that the value on a wire in a network is the same irrespective of the direction from which it is computed). Furthermore, by looking at the mathematical definition above and setting $v_1 = \alpha_i$ and $v_2 = \beta_i$ we have that $\alpha_{i+1} = w_1 = \beta_i$ and $\beta_{i+1} = w_2 = v_1 \oplus f_i(v_2) = \alpha_i \oplus f_i(\beta_i)$. However, since $\alpha_{i+1} = \beta_i$ this is the same as saying that $\alpha_i = \beta_{i+1} \oplus f_i(\alpha_{i+1})$, fulfilling Equation (5.1). Of course, the important observation in Equation (5.1) is that α_i and β_i can be efficiently computed from α_{i+1} and β_{i+1}, as long as f_i is efficiently computable. Given that this is possible in any round, we have that all of the rounds of the network can be inverted, thus yielding an efficient procedure for inverting the entire block cipher. We therefore have the following claim:

CLAIM 5.2 *In a Feistel network F in which the f-function can be efficiently computed, there exists an efficient procedure for computing $F^{-1}(y)$. Furthermore, for every key k and every input x, $F_k^{-1}(F_k(x)) = x$.*

We remark that typically the f-function in a Feistel network is constructed from S-boxes and mixing permutations, exactly as in a substitution-permutation network. The main difference is thus that in a Feistel network, the S-boxes need not be invertible. When this is indeed the case (i.e., the S-boxes and mixing permutations are used in a similar way), attacks on Feistel networks with very few rounds can be designed in the same way as described above for substitution-permutation networks.

5.3 DES – The Data Encryption Standard

The Data Encryption Standard, or DES, was developed in the 1970s at IBM (with some help from the National Security Agency), and adopted in 1976 as a Federal Information Processing Standard (FIPS) for the US. In its basic form, DES is no longer secure due to its short key size. Nevertheless, it is still widely in use today in the form of triple DES (triple DES is a block cipher that is based on DES and is described below in Section 5.4). The DES block cipher has undergone great scrutiny; arguably more than any other encryption algorithm in history. The common consensus is that it is extremely secure. Indeed, the best known attack on DES in practice is a brute force (or exhaustive key) search on its key space (i.e., attack by trying all possible keys and seeing which key decrypts the message correctly). As we will see below, there are important theoretical attacks on DES that require less computation than such a brute force attack. However, they provide no real threat in practice. The DES algorithm was recently replaced by the Advanced Encryption Standard (AES). Nevertheless, as we have mentioned, it is still widely used in the form of triple DES. In this section, we will describe the details of the DES construction. We stress that we will not provide the full specification, and some parts of the design will be omitted from our description (e.g., the permutation on the key before the key schedule). Rather, our aim is to present the basic ideas behind the construction.

5.3.1 The Design of DES

The DES block cipher is a 16-round Feistel network with a block size of 64 bits and a key length of 56 bits. Recall that in a Feistel network the internal f-function works on half a block at a time. Thus, the input and output length of the DES internal f-function is 32 bits. Furthermore, as is to be expected, the

DES f-function is *non-invertible*, thereby utilizing the advantage of a Feistel network over a substitution-permutation network. In each of the 16 rounds of DES the same internal f-function is used. However, a different subkey is derived in each round. Each subkey is of length 48 bits, and is composed of a subset of the 56 bits of the master key (in each round, a different subset is chosen and so there is a different subkey). The way the subkeys are chosen is called the key schedule. We will not show exactly how the key schedule works. Rather, it suffices for us to note that an initial permutation is applied to the bits of the key. Following this, it is divided into two halves of length 28 bits each. Then, in each round, the left 24 bits of the subkey are taken as a subset of the left 28 bits in the (permuted) master key, and the right 24 bits of the subkey are taken as a subset of the right 28 bits in the (permuted) master key. We stress that the initial permutation and the choice of which subsets are taken in each round are fixed and public. The only secret is the key itself.

We note that before and after the Feistel network, DES actually applies a *fixed and known* permutation to the input and its inverse to the output; this permutation is known as the *initial permutation* and is denoted IP. This slows down software implementations of DES (the computation of this permutation takes about one third of the running time, in contrast to hardware where it takes almost no time). As with the permutation on the key, we will ignore these permutations in our description and analysis below because they play no security role beyond slowing down attackers who use software.

The internal DES f-function. The f-function is constructed using the same basic building blocks of S-boxes and mixing permutations described above for substitution-permutation networks. The exact construction works as follows. The first step in the f-function is to *mix* the 48-bit subkey with the 32-bit input. This mixing is carried out via a bitwise exclusive-or. However, this operation works on two strings of the same size. Thus, the 32-bit input to the f-function is first passed through an expansion function $E : \{0,1\}^{32} \rightarrow \{0,1\}^{48}$ that takes inputs of size 32 and produces outputs of size 48. The expansion is very simple and works by just duplicating half of the input bits. The 48-bit output of the expansion function is then exclusively-ored with the subkey. In summary, the first step in the f-function is to compute $E(x) \oplus k$, where x is the 32-bit input and k is the 48-bit subkey of the round.

This intermediate 48-bit result is then divided into 8 blocks of size 6 bits each. Each block is passed through an S-box that takes inputs of length 6 bits and produces outputs of length 4 bits. There are 8 S-boxes, denoted $S_1, \ldots, S_8$. Notice that the S-boxes are *not* invertible because the output is shorter than the input. This is the non-invertible part of the DES internal f-function. As with the S-boxes in a substitution-permutation network, these small functions (lookup tables) have non-linear and random behavior and provide the "confusion" portion of the block cipher. Notice that the output of the S-box computations consists of 8 blocks, each of length 4 bits. Thus, we obtain an intermediate value of length 32 bits; exactly the size of the output

of the f-box. The last step in the computation is the "diffusion" step. That is, the outputs from the S-boxes are all passed through a mixing permutation. We stress once again that the expansion function E, the lookup tables defining the S-boxes and the final mixing permutation are all known and fixed. The only unknown value is the master key. See Figure **??** for a diagram of the construction.

The *S*-boxes. The definition of the S-boxes is a crucial element of the DES construction. In fact, even though the DES S-boxes look very random, they were very carefully designed (reportedly, with the help of the National Security Agency). Studies on DES have shown that if the S-boxes are chosen truly at random, then DES becomes much easier to break. These attacks use advanced cryptanalytic techniques called differential cryptanalysis (see below for a brief description). 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 render the entire construction insecure. Due to their importance, we will describe some basic properties of the DES S-boxes:

1. Each box can be described as a table with 4 row and 16 columns, where each entry in the table contains 4 bits. (The 64 entries correspond to the 2^6 possible inputs of length 6 bits.)
2. The first and last input bits are used to choose the *table row* and bits 2 to 5 are used to choose the *table column*.
3. Each row in the table is a *permutation* of $0, 1, 2, \ldots, 15$.
4. Changing *one input bit*, always changes at least *two output bits*.

We will use some of the above properties in our analysis of reduced-round DES below.

The DES avalanche effect. As we have discussed above in the context of substitution-permutation networks, the avalanche effect is one of the crucial properties of any secure block cipher (i.e., pseudorandom permutation). The fourth property of the DES S-boxes described above ensures that DES has 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. If the difference between the inputs is in the half of the input that does not enter the internal f-function in the first round, then after this round the intermediate values still differ by only a single bit (note that the other half of the input is the same and so it remains the same after running it through f). Now, in the second round of DES, the half of the input with the single-bit difference is run through the internal f-function. Assuming that the bit is not duplicated in the expansion function, we have that the intermediate values before applying the S-boxes still differ by a single bit. By property 4 above, we have that the intermediate values after the S-box computation differ in at least *two* bits. The mixing permutation then spreads

these two bits into different areas of the block. In the next third round, each of these two bits enters a different S-box and so by following the same arguments as above, the results differ in at least 4 bits. As with a substitution-permutation network we have an exponential avalanche and so after the 4^{th} round the values differ in 8 bits, after the 5^{th} round the difference is 16 bits and after the 6^{th} round the difference is 32 bits, thereby completing the avalanche. Notice that DES has 16 rounds, and so the avalanche is completed very early on in the computation. This ensures that the computation of DES on similar inputs yields completely different and independent-looking outputs. We remark that the success of the DES avalanche effect is also due to the choice of the mixing permutation. In fact, it has been shown that a random mixing permutation yields a far weaker avalanche effect, and results in DES being much easier to break.

The DES standard. A complete description of the DES standard can be obtained from the FIPS (Federal Information Processing Standards Publications) website of NIST (National Institute of Standards and Technology).

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 its behavior with only a few rounds. We will look at DES with one, two and three rounds (recall that the real DES has 16 rounds). Some of the attacks are similar to those discussed for substitution-permutation networks. However, here we will see how they are applied to a concrete block cipher rather than to a general design. In the attacks below, we will show how to obtain the secret key from input/output pairs. Clearly DES with 3 rounds or less cannot be a pseudorandom function because the avalanche effect is not yet completed in 3 rounds (exactly as in the attack on a three-round Substitution-Permutation Network). Thus, our aim is to show something much stronger; i.e., that the secret key can be fully determined.

In all of the attacks below, we assume that the adversary has a number of pairs (x, y) where $y = DES_k(x)$, and k is the secret key. (We will not need the fact that the adversary can adaptively choose the values x for which it will receive $y = DES_k(x)$). Since DES is a Feistel network, the internal f-function works on half a block at a time. We denote the left half of the input by x_1 and the right half by x_2. Likewise, we denote the left half of the output by y_1 and the right half of the output by y_2.

Single-round DES. In a single round of DES, we have that $y_1 = x_2$ and $y_2 = x_1 \oplus f_1(x_2)$ where f_1 is the internal f-function in the first and only round (see the definition of a Feistel network above). We therefore know the complete input and output to f_1. Specifically, the input to f_1 is x_2 and the output is $y_2 \oplus x_1$. The first step is to apply the inverse of the mixing permutation to the output. This yields the intermediate value that contains the outputs from all the S-boxes, where the first 4 bits are the output from

S_1, the next 4 bits are the output from S_2 and so on. This means that we have the exact output of each S-box. As we have seen, each row of an S-box is a permutation over $0, \ldots, 15$ in binary form. Therefore, given the output of an S-box, the only uncertainty remaining is from which row it came. Stated differently, each value appears only once in each row. Thus, given the output value, the row number completely determines the 6-bit input to the S-box. Since there are 4 rows, it follows that there are only 4 possible input values that could have occurred. Each of these possible input values is the XOR of $E(x_2)$ with the 48-bit key (where E is the expansion function). Now, since x_2 is known, we conclude that for each 6-bit portion of the 48-bit key, there are 4 possible values. This is true for each of the 8 S-boxes, and so we have reduced the possible number of keys from 2^{48} to $4^8 = 2^{16}$ (because each of the 8 portions have 4 possibilities). This is already a very small number and so we can just try all of the possibilities on a different input/output pair (x', y'). We therefore obtain the full key in very little time.

Two-round DES. In two rounds of DES, with internal functions f_1 and f_2 in the first and second rounds, respectively, we have the following:

1. The input to f_1 is x_2 and the output is $y_1 \oplus x_1$. (This follows because in the Feistel computation with two rounds, the output $f_1(x_2)$ is XORed with x_1 and then appears as the left half of the output.)

2. The input to f_2 is y_1 and the output is $y_2 \oplus x_2$.

(We strongly recommend drawing a 2-round Feistel network in order to verify the above.) We therefore know the inputs and outputs the both f_1 and f_2. Thus, the same method can be used as above for single-round DES. Note that this attack works even if a completely different 48-bit key is used in each round.

Three-round DES. See Figure **??** for a diagram of DES reduced to only three rounds. In order to describe the attack, we have denoted the intermediate values on the wires during the computation, as in Figure **??**. As usual, the input string to the cipher is denoted $x = (x_1, x_2)$ and the output string is denoted $y = (y_1, y_2)$. Note that $\alpha_1 = x_2$ and $\beta_2 = y_1$. Thus the only unknown values amongst $\alpha_1, \alpha_2, \beta_1, \beta_2$ are α_2 and β_1 (where in fact $\alpha_2 = \beta_1$).

In the case of 3-round DES, we do not have the input and output of the internal f-function in each round. For example, let us consider f_2. In this case, we know the output because tracing the wires we can see that it equals $\alpha_1 \oplus \beta_2$, where as we have mentioned $\alpha_1 = x_2$ and $\beta_2 = y_1$. Thus, the output of f_2 is $x_2 \oplus y_1$ and so is known. In contrast, we do *not* know the input value to f_2. By tracing the wires we see that the input to f_2 equals $x_1 \oplus f_1(x_2)$ or equivalently $y_2 \oplus f_3(y_1)$, but neither of these values are known. (Note that we can trace the wires in either direction and this makes no difference.) A similar exercise yields that for both f_1 and f_3 we know the inputs, but not the outputs. Thus, the attack that we used to break DES with one and

two rounds will not work here. Rather, instead of relying on full knowledge of the input and output of one of the f-functions, we will use knowledge of a certain relation between the inputs and outputs of f_1 and f_3. We begin by describing the relation that we will use. Observe that the output of f_1 equals $x_1 \oplus \alpha_2 = x_1 \oplus \beta_1$. Furthermore, the output of f_3 equals $\beta_1 \oplus y_2$. Taking the XOR of the output of f_1 with the output of f_3 we obtain the value $(x_1 \oplus \beta_1) \oplus (\beta_1 \oplus y_2) = x_1 \oplus y_2$. Since both x_1 and y_2 are known, we have that the *exclusive-or of the outputs of* f_1 *and* f_3 is known. Furthermore, the input to f_1 is x_2 (and so is known) and the input to f_3 is y_1 (and so is known). We conclude that from the overall input and output we can determine the inputs to the internal functions 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 in DES, the subkeys in each round are generated by rotating each half of the key (we are ignoring the initial permutation on the key, which makes no difference to the security). This means that the left half of the key affects the S-boxes $S_1, \ldots, S_4$ only, and the right half of the key affects the S-boxes $S_5, \ldots, S_8$ only. Since the permutation after the S-boxes is fixed, we also know which bits come out of which S-box.

Since the length of the entire key is 56 bits, it follows that there are 2^{28} possible half-keys (for each half). The idea behind the attack is to separately traverse the key-space for each half of the key. If we can verify the guess of a half key, then this is possible. Now, let k_L be a guess for the left half of the key. We know the input x_2 into f_1 and so using the guess k_L as the key and x_2 as the input, we can compute the output of $S_1, \ldots, S_4$ in f_1. This implies that we can compute half of the output of f_1 (the mixing permutation spreads out the 16 computed bits, but we know exactly which bits these are). Likewise, we can compute the same locations for the output of f_3 by using input y_1 and the same guessed half-key k_L. Finally, we can compute the XOR of these output values and see if they match the appropriate bits in $x_1 \oplus y_2$ which is known (recall that $x_1 \oplus y_2$ equals the XOR of the outputs of f_1 and f_3). If they are not equal, then we know that the guessed half key is incorrect. If they are equal, then we take the half-key as a candidate. We stress that there are likely to be many candidates. In particular, since we consider 16 bits of output, an incorrect key is accepted with probability approximately 2^{-16} (assuming random behavior of DES). There are 2^{28} keys and so approximately 2^{12} keys are expected to cause equality and thus be accepted as candidates for the left half of the key.

The above method is carried out separately for each half of the key. The result is that in time $2 \cdot 2^{28}$ we obtain 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 remain with 2^{24} candidate keys overall and can run a brute-force search over them all. The total complexity of the attack is $2^{28} + 2^{28} + 2^{24}$ which is less than 2^{30}. We remark that an attack of complexity 2^{30} can easily be carried out on standard personal computers.

5.3.3 The Security of DES

As we discussed in Section 1.3, no encryption scheme can be secure if it is possible to traverse the entire key space. This is due to the fact that it is possible to try to decrypt a ciphertext using all possible keys. If the ciphertext is long enough (e.g., longer than the key), then with high probability only one key will map the given plaintext to the ciphertext. (Note, of course, that multiple plaintext/ciphertext pairs can be used to achieve the effect of a "long enough" ciphertext.) The same observation is true of block ciphers (as pseudorandom permutations and a building block for encryption schemes). Specifically, in the setting of pseudorandom permutations the adversary can query its oracle (see Definition 3.28) and obtain a number of pairs (x, y) where y is the output of the block cipher on input x (and a secret key). Thus, given enough time, the adversary can always determine the secret key by finding the unique key that maps the inputs to their respective outputs, as obtained from its oracle. Such an attack is called a *brute force attack* or *exhaustive key search* and it can be carried out on any block cipher. We therefore conclude that since DES has a 56-bit key, it can be broken in time 2^{56}. In fact, due to the "complementary property" of DES (see Exercise ***), it can actually be broken in time 2^{55}. Although this is not a trivial amount of computation, it is definitely feasible today. In fact, already in the late 1970's there were strong objections to the choice of such a short key for DES. Back then, the objection was more theoretical as no one had the computational power to traverse that many keys. The practicality of a brute force attack on DES was demonstrated in 1997 and 1998 when a number of DES challenges set up by RSA Security were solved (these challenges were input/output pairs and a reward was given to the first person or organization to find the secret key that was used to compute the output). The first challenge was broken in 96 days in 1997 by the DESCHALL project. The second challenge was broken in early 1998 in 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* (it was built by the Electronic Frontier Foundation at a cost of $250,000). Additional challenges have been solved, and the latest was solved in just 22 hours and 15 minutes (as a combined effort of Deep Crack and distributed.net). The bottom line of the above is that DES is no longer secure, and in fact has not been secure in a very long time (even an attack taking a full year is completely unacceptable).

It is important to note 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. Thus, it makes sense to try to use DES as a building block in order to construct a block cipher with a longer key. This was carried out very successfully and the result is called triple DES. We discuss this construction below in Section 5.4.

One issue worth noting with respect to brute force attacks on DES is that given a single input/output pair (x, y), with probability approximately 2^{-8}

there may be more than one key that maps x to y. This occurs because for a single randomly-chosen key k, the probability that $DES_k(x) = y$ is 2^{-64} (assuming random behavior of DES). Thus, the probability that there exists a key k (that is not equal to the real key used to compute y from x) may be close to $2^{56}/2^{64} = 2^{-8}$. This means that with a not too small probability, a brute force search may turn up more than one candidate key. In such a case, it is easy to rule out the additional candidate or candidates by trying another pair (x', y'); note that it suffices to test the few remaining candidate keys on the additional input/output pair.

5.3.3.1 Advanced Cryptanalytic Attacks on DES

The brute force attacks described above do not utilize any internal weaknesses of DES (apart from the complementary property that reduces the attack time by one half). 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 1980s who developed a technique called *differential cryptanalysis* and used it to achieve an attack on DES that took less time than 2^{55}. Their specific attack takes time 2^{37} and works by analyzing 2^{36} outputs that are obtained from a larger pool of 2^{47} chosen inputs. Thus, in essence 2^{47} time is really needed. Nevertheless, the real problem is that the adversary needs to be able to make 2^{47} queries to its oracle (or in real-life, it needs to be able to make something like 2^{47} chosen-plaintext requests). It is hard to imagine any realistic scenario where such a large amount of chosen inputs can be obtained. We stress that this does not take away from the importance of the work of Biham and Shamir. However, it does mean that as an attack on DES, it is more theoretical than practical. We note that it is believed that differential cryptanalysis as a technique was already known to the designers of DES at the time that DES was developed. One of the reasons for this belief is the fact that Biham and Shamir also showed that a variant of DES with random S-boxes is much easier to break using differential cryptanalysis than the actual DES. Later, the designers of DES at revealed that the method was indeed known to them (but that they were asked to keep it quiet in the interest of National security). Thus, the general belief is that the actual S-boxes of DES were specifically chosen to thwart differential cryptanalytic attacks, and thus the NSA already knew about these attacks at the time.

Following Biham and Shamir's breakthrough, an additional cryptanalytic attack called *linear cryptanalysis* was developed by Matsui in the early 1990s and applied to DES. The advantage of Matsui's attack is that although it still requires a large number of outputs (2^{43} to be exact), it suffices for them to be known-input only. That is, the adversary needs to be able to obtain 2^{43} input/output pairs. This is a great improvement over the requirement that the adversary needs to be able to also choose the inputs to the block cipher. Nevertheless, it is still hard to conceive of any real scenario where it is possible to obtain such a large number of input/output (or plaintext/ciphertext) pairs.

We conclude that although using sophisticated cryptanalytic techniques it is possible to break DES in less time than required by a brute-force attack, in practice exhaustive key search is still the most effective attack on DES today. 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). In Section 5.6 we will briefly describe the basic ideas behind differential and linear cryptanalysis.

5.4 Increasing the Key Size for Block Ciphers

As we have seen, the DES design seems to be almost optimal. It has withstood decades of cryptanalytic attacks and an exhaustive key search still remains the best attack in practice. Thus, it is very natural to try to build a block cipher with a long key, using DES as a building block. In this section we will study such constructions.

Internal tampering versus black-box constructions. There are two possible approaches that one could take to this task. The first is to somehow try to modify the internal structure of DES, while increasing the key size. For example, one may leave the internal f-functions untouched and simply use a 128-bit key with a different key schedule (still choosing a 48-bit subkey in each round). The disadvantage of this approach is that by modifying the design of DES we lose the confidence that we have gained over the many years of its existence. Cryptographic constructions are very sensitive and even mild and seemingly insignificant changes can render the original scheme completely insecure. This approach is therefore usually not recommended. An alternative approach that does not suffer from the above problem is to use the original DES as a "black box". That is, no change is made to the original cipher, and the key is lengthened by somehow applying the complete original DES a number of times to the input (while using different keys each time). For example, in double-DES, the block cipher is defined by two applications of DES to the input where each application uses an independent key. Another advantage of black-box constructions is that they can be applied to any underlying block cipher, because they make no reference to the internal structure. Indeed, in the constructions below we will often refer to an arbitrary block-cipher, and not just to DES.

5.4.0.2 Double Invocation

Let F be a block cipher and let k_1 and k_2 be two independent keys for F. Then, a new block cipher with a key that is twice the length of the original

one can be defined by

$$F'_{k_1,k_2}(x) = F_{k_2}(F_{k_1}(x)).$$

If F =DES then the result is a key of size 112, which is much too long for any exhaustive key search (for DES this method is typically called double-DES). Unfortunately, as we will show now, a double invocation of a block cipher does not provide a high enough level of security. We describe a "meet-in-the-middle" attack on the double-invocation method. Denote the length of the keys of F by n (thus the length of the keys of F' is $2n$). The attack that we will describe now uses approximately 2^n time and 2^n space.

The adversary is given an input/output pair (x, y) where $y = F'_{k_1,k_2}(x) = F_{k_2}(F_{k_1}(x))$, and works as follows. First, it starts by building two lists of pairs. The first list is made up of all the pairs of the form $(\tilde{k}_1, z_1)$ where $z_1 = F_{\tilde{k}_1}(x)$; that is, for every possible key $\tilde{k}_1$, the pair $(\tilde{k}_1, z_1)$ is added to the list. The second list is made up of all the pairs of the form $(\tilde{k}_2, z_2)$ where $z_2 = F^{-1}_{\tilde{k}_2}(y)$. (Recall that x and y are the given input/output pair.) Notice now that there exists a value z such that $F_{k_1}(x) = z = F^{-1}_{k_2}(y)$, where k_1 and k_2 are the keys that the adversary is searching for. Therefore, the aim of the adversary is to match up pairs in the first list with pairs in the second list, where a match is defined by the pairs having the same z portion (i.e., where $z_1 = z_2$). Any such match defines a candidate key $(\tilde{k}_1, \tilde{k}_2)$ for F' because $F_{\tilde{k}_1}(x) = z_1 = z = z_2 = F^{-1}_{\tilde{k}_2}(y)$ and so $y = F_{\tilde{k}_2}(F_{\tilde{k}_1}(x))$. This is the "meet-in-the-middle" that we are looking for; see Figure **??**.

We therefore remain with an algorithmic problem which is to scan the two lists and find all matches. We leave the solution of this as an exercise (see Exercise 5.8), and note that it can be carried out in time $\mathcal{O}(2^n)$.

Assuming random behavior of F, we have that approximately 2^n candidates key-pairs $(\tilde{k}_1, \tilde{k}_2)$ should be chosen. (This is because each z should appear approximately once in each table. Thus, each z in the first table should have approximately one match in the second table. This yields 2^n candidates.) The attack is then concluded by testing all of the candidate pairs on a new input/output pair (x', y') obtained by the adversary.

Complexity. Using counting sort, the lists can be constructed and sorted in time $\mathcal{O}(2^n)$. Furthermore, the search for all candidates can be carried out in time $2 \cdot 2^n$. Overall, the time complexity of the attack is therefore $\mathcal{O}(2^n)$. In other words, double-invocation is vulnerable to an attack that takes no longer than an exhaustive key search on the original block cipher. We stress that although this is true with respect to the time complexity of the attack, it requires $2 \cdot 2^n$ memory, which is a very high space complexity.

Double-DES. When applying double invocation to DES, we obtain that the result is vulnerable to an attack requiring time that is in the order of 2^{56} (to be more exact, it would be something like 2^{60}). This is still within our computing capabilities today and so is highly problematic. Of course, the

attack also requires 2^{57} space and this is far more problematic. Despite this, the margin of security for double-DES is not large enough and it is therefore not used.

5.4.0.3 Triple Invocation

In order to thwart meet-in-the-middle attacks, three invocations of the underlying block cipher can be used. We have no proof that no other shortcuts exist for this method. Despite this, it is widely believed that triple invocation of the block cipher provides a high level of security against brute force attacks. There are two variants that are typically used for triple invocation:

1. *Variant 1 – three independent keys:* Choose 3 independent keys k_1, k_2, k_3 and compute $y = F'_{k_1,k_2,k_3}(x) = F_{k_3}(F^{-1}_{k_2}(F_{k_1}(x)))$.

2. *Variant 2 – two independent keys:* Choose 2 independent keys k_1, k_2 and compute $y = F'_{k_1,k_2}(x) = F_{k_1}(F^{-1}_{k_2}(F_{k_1}(x)))$.

Before comparing the security of the two alternatives we note that the middle invocation of F is actually F^{-1}. This makes no difference to the security because if F is a pseudorandom permutation then so too is F^{-1} (see Definition 3.28). 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 helps with backward compatibility (in order to reverse back to a single invocation, there is no need for special code and it suffices to just set the keys to all be equal). Regarding the security of the alternatives, no weakness whatsoever has been demonstrated with the first alternative. In contrast, it has been shown that for the second alternative it is possible to carry out an attack in time 2^n and using 2^n queries to the block cipher. However, as we have mentioned above, the possibility of obtaining 2^n outputs of the block cipher on chosen inputs is so far fetched that it is not considered a concern at all. (This is in contrast with 2^n memory that is far more feasible.) Therefore, both alternatives are reasonable, although it seems preferable to go with the first.

Triple-DES (3DES). Triple-DES is based on a triple invocation of DES, as described above. It is widely believed to be highly secure and in 1999 officially replaced DES as the NIST standard (although one would hope that the basic DES was already phased out well before this time). We remark that triple-DES is still widely used today and is considered a very strong block cipher. Its only drawbacks are its relatively small block-size (that can be problematic as discussed in the paragraph on "block length and security" in Section 3.6.4) and the fact that it is quite slow since it requires 3 full block cipher operations (in fact, even single DES is not that fast, making triple DES even worse). These drawbacks have led to its recent replacement in 2001 by the Advanced Encryption Standard (AES), presented in the next section.

5.5 AES – The Advanced Encryption Standard

In January 1997, the National Institute of Standards and Technology of the United States (NIST) announced that they were seeking a new block cipher to replace the DES standard. The new cipher was to be called the *Advanced Encryption Standard*, or AES for short. Later that year the terms of the AES competition were published. The competition was ingeniously designed and resulted in extraordinarily intensive scrutiny on the proposed ciphers. This was achieved by having two rounds in the competition. In the first round, any team could submit a *candidate algorithm*. There were 15 different algorithms that were submitted from all over the world. These submissions included the work of many of the best cryptographers and cryptanalysts today. Following the first round of submission, the different algorithms were analyzed for their security and performance. Two AES conferences were held, one in 1998 and one in 1999, in which papers were published regarding the different security and other properties of the submitted schemes. Following the second AES conference, NIST narrowed the field down to 5 submissions and the second round begun. A third AES conference was then held, inviting additional scrutiny on the five finalists. Finally, in October 2000 NIST announced that the winning algorithm is Rijndael (a block cipher designed by John Daemen and Vincent Rijmen from Belgium). This process was ingenious because any group who submitted an algorithm, and was therefore interested in having their algorithm be adopted, had strong motivation to attack all the other submissions.[4] In this way, essentially all of the world's best cryptanalysts worked intensively to find even the slightest weaknesses in the AES submissions. Thus, after only a few years, the scrutiny received was very great, and our confidence in the security of the winning algorithm was high. Of course, the longer the algorithm is used, the more our confidence will grow. However today, only about 5 years later, the AES block cipher 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 the AES block cipher. As with DES, we will not present a full specification and our description should never be used as a basis for implementation. In particular, we will not present the AES S-box or describe the key schedule. Rather, our aim is to provide a general idea of how the algorithm works. Before we begin, we remark that although the terms AES and Rijndael are used interchangeably, the AES is a specific standard and is different from Rijndael. For example, Rijndael can be implemented with a large range of

[4] We note that the motivation was not financial because the winning submission could not be patented. Nevertheless, much honor and glory was at stake.

block and key sizes whereas the AES is limited to a block size of 128 and a key size of 128, 192 or 256 bits.

In contrast to DES that has a Feistel structure, AES is essentially a substitution-permutation network. The AES algorithm holds a 4 by 4 array of bytes called the *state*, that is initialized to the input to the cipher (note that the input is 128 bits which is exactly 16 bytes). The substitution and permutation operations (providing confusion and diffusion) are all applied to the state array. There are four stages in every round of AES (see Figure **??** for a graphical presentation of each of the four steps):

1. *Stage 1* – AddRoundKey*:* In every round of AES, a 16 byte round key is derived from the master key, and is interpreted as a 4 by 4 array of bytes.[5] Then, the key array is simply XORed with the state array. Denote by $a_{i,j}$ the byte appearing in the i^{th} row and j^{th} column of the state array, and likewise by $k_{i,j}$ that analogous byte in the key array. Then the AddRoundKey step consists of computing $a_{i,j} = a_{i,j} \oplus k_{i,j}$ for every $1 \leq i \leq 4$ and $1 \leq j \leq 4$.

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 $\{0,1\}^8$. Thus, the SubBytes step consists of computing $a_{i,j} = S(a_{i,j})$ for every $1 \leq i \leq 4$ and $1 \leq j \leq 4$. We stress that there is only *one* S-box and it is used for substituting *all* of the bytes in the state array.

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. Of course, all shifts are cyclic so in the second row, we have that byte $a_{2,1}$ become $a_{2,4}$, byte $a_{2,2}$ becomes $a_{2,1}$ and so on.

4. *Stage 4* – MixColumns*:* In this step, each column is mixed via an invertible linear transformation. Specifically, each column is interpreted as a polynomial over $GF[2^8]$ (with the entries being the polynomial coefficients) and is multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$. This step can also be viewed as a matrix multiplication in $GF[2^8]$.

By viewing stages 3 and 4 as a "mixing permutation" step, we have that each round of AES has the structure of a substitution-permutation network. (That is, the round key is first XORed with the intermediate value. Then,

[5] Recall that the master key may be of size 16, 24 or 32 bytes. This effects the key schedule only.

confusion is applied by applying a small, invertible random-looking function to each byte. Finally, the bytes are mixed. We stress that unlike our general description of a substitution-permutation network, here the third stage is not via a simple mixing permutation.)

The number of rounds in AES depends on the key-size. 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 simply rewinding the last round as we have seen).

Security of AES. As we have mentioned, the AES cipher underwent intense 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 they vary on different parameters. We will therefore describe the complexity of one set of attacks that gives the flavor of what is known. There are known attacks on 6-round AES for 128 bit keys (taking time in the order of 2^{72} encryptions), 8-round AES for 192 bit keys (taking time in the order of 2^{188} encryptions), and 8-round AES for 256 bit keys (taking time in the order of 2^{204} encryptions). We stress that the above attacks are for *reduced-round* variants of AES, and as of today nothing better than an exhaustive search on the key space is known for the full AES construction. (Observe that even the complexities on the reduced-round variants are very high.) Of course, the fact that no attack is known today does not mean that one does not exist. In any case, it appears that AES has a large security margin (especially if one uses 192-bit or 256-bit keys).

A different class of attacks, called *side-channel attacks*, consider the scenario that the adversary is somehow able to make measurements on the processor computing the encryption or decryption with the unknown secret key. Although it sounds far-fetched, information such as the time taken to encrypt or the power consumed during encryption has actually been used to extract secret keys in reasonable amounts of time. Such attacks are sometimes feasible (e.g., when a secure coprocessor, typically called a smartcard, is used for carrying out encryption and decryption, and may fall into the hands of the adversary). In any case, a number of side-channel attacks have been demonstrated on AES. However, they typically require very strong assumptions on the power of the adversary.

We conclude that as of today, AES constitutes an excellent choice for almost any cryptographic implementation that needs a pseudorandom permutation or function. It is free, standardized 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 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 (and even many experts) should not try to construct new ciphers unless there is a very good reason to. Given that we have 3DES and AES, in most applications it is hard to justify the necessity to use anything else.

In this section we will very briefly mention two central tools that belong in the cryptanalysts toolbox. The existence of such tools should also help to strengthen our above warning that it is very hard to construct good block ciphers. Today, any new cipher must demonstrate resilience to differential and linear cryptanalysis.

Differential cryptanalysis. This technique was first presented by Biham and Shamir in the late 1980s who used it to attack DES. The basic idea behind the attack is to find *specific differences in input* that lead to *specific differences in output* with above expected probability. Let x_1 and x_2 be two inputs to the block cipher. The difference between x_1 and x_2 is defined by $\Delta_x \stackrel{\text{def}}{=} x_1 \oplus x_2$. Likewise, let y_1 and y_2 be the output of the block cipher with a secret key k, given inputs x_1 and x_2, respectively (i.e., $y_1 = F_k(x_1)$ and $y_2 = F_k(x_2)$), and let $\Delta_y \stackrel{\text{def}}{=} y_1 \oplus y_2$. The pair (Δ_x, Δ_y) is called a differential.

Differential cryptanalysis capitalizes on a potential weakness in the cipher that results in certain differentials appearing with probability that is higher than expected in a random function. In order to clarify, we say that a differential (Δ_x, Δ_y) appears with probability p if for random plaintexts x_1 and x_2 such that $x_1 \oplus x_2 = \Delta_x$, the probability that $F_k(x_1) \oplus F_k(x_2) = \Delta_y$ is p. Denoting now the size of the block by n, it is clear that in a random function, no differential should appear with probability that is much higher than 2^{-n}. However, in a block cipher (especially a weak one), there are differential that appear with significantly higher probability.

We will not discuss here how one finds such differentials or even how they are used to extract the secret key. We will 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 that is less than a brute force search. Notice that differential cryptanalysis uses a *chosen-plaintext attack*. This can be seen from the fact that the method works by observing what happens with pairs

of plaintexts with a given differential (in order to observe pairs with this differential, the attacker must be able to choose the pairs of plaintexts it wishes). Thus, if an attack requires a large number of plaintext pairs, its practicality is in question.

We remark that although DES and AES are resilient to differential cryptanalysis (or at least, they seem to be), it has been used with success on 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 1990s. Matsui's method works by considering linear relationships between some of the bits of input and output. That is, the potential weakness here is that some subset of the plaintext and ciphertext bits are linearly correlated. Letting $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ denote the input and output bits, respectively, linear cryptanalysis considers equations of the form $x_{i_1} \oplus \cdots x_{i_\ell} \oplus y_{i_1} \oplus \cdots y_{i'_\ell} = 0$. Clearly, if the function in question was truly random, then such linear equations can only hold with probability 1/2. Thus, here the cryptanalyst looks for equations of the above type that hold with probability that is far from 1/2. Matsui showed how to use such equations to completely break the cipher by finding the secret key. However, we note that the existence of such an equation that holds with probability that is *non-negligibly* far from 1/2 is enough to declare that the block cipher is not pseudorandom. (Specifically, any such equation holding with probability $1/2 + \varepsilon$ yields an algorithm that can distinguish the block cipher from a random function with advantage ε. All the algorithm needs to do is to output 1 when the equation holds and 0 otherwise.) An important property of this attack is that it does not require chosen-plaintext abilities. Rather it suffices for the cryptanalyst to have many plaintext/ciphertext pairs (i.e., here the cryptanalyst carries out a known-plaintext attack). Despite this, as with differential cryptanalysis, if a very large number of pairs are needed the attack becomes impractical in most settings.

We have already discussed the applications of differential and linear cryptanalysis on DES in Section 5.3.3 and therefore do not repeat it here.

5.7 Stream Ciphers from Block Ciphers

In this chapter we have studied practical constructions of block ciphers. We have not covered stream ciphers, and will not do so in this book. There are a number of stream ciphers in use today; just one popular example is RC4. As we have mentioned (in Section 3.4.3), it seems that the cryptographic community's understanding of stream ciphers is somewhat less satisfactory than its understanding of block ciphers. This can be seen by the fact that extremely

strong block ciphers like 3DES and AES exist and have been standardized. In contrast, stream ciphers seem to be far more prone to attack, and there is no standard stream cipher with no known weakness that has withstood years of cryptanalytic attacks.

Having said the above, it is well known that stream ciphers can easily be constructed from block ciphers, in which case the stream cipher inherits the security of the block cipher. We have already seen how this can be achieved when we studied modes of operation for block ciphers (see Section 3.6.4). Therefore, unless severe constraints mandate the use of a dedicated stream cipher (e.g., in the case of weak hardware where the additional efficiency of stream ciphers is crucial), we advocate the use of AES and 3DES in practice. We note that AES is extremely fast and so for most applications it more than suffices.[6]

Additional Reading and References

The confusion-diffusion paradigm and substitution-permutation networks were both introduced by Shannon [113]. The Feistel method of constructing a block cipher was presented by Feistel [54] when working on Lucifer, a block cipher predating DES. A theoretical analysis of the Feistel methodology was later given by Luby and Rackoff [91].

The full DES standard can be found at [102] and a more friendly description can be found in Kaufman et al. [84]. The most comprehensive presentation of AES can be found in the book written by its designers Daemen and Rijmen [44]. 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 [93, Chapter 7].

A recent analysis of the security of triple-DES is given by Bellare and Rogaway [22].

Differential cryptanalysis was introduced by Biham and Shamir [23] and its use on full DES was presented in [24]. Coppersmith [37] describes the DES design in light of the public discovery of differential cryptanalysis. Linear cryptanalysis was discovered by Matsui [92]. For more information on these advanced techniques, we refer to the excellent tutorial on differential and

[6] One may wonder why stream ciphers are preferred over block ciphers in the first place. There are two main reasons. First, they are often faster (and our recommendation to use a block cipher in a "stream-cipher mode" will therefore cause some slowdown). The second reason is that implementors often prefer to use a stream cipher because no padding of the message is needed (recall that when encrypting with a block cipher, the plaintext message must be padded in order to make its length an exact multiple of the block size).

linear cryptanalysis by Heys [78]. A more concise presentation can be found in the textbook by Stinson [124].

Exercises

5.1 In our attack on a two-round substitution-permutation network, we considered a block size of 64 bits and a network with 16 S-boxes that take input of size 4 bits. Repeat the analysis for the case of 8 S-boxes, each S-box taking an input of size 8 bits. Present an exact description of the complexity of the attack. Repeat the analysis again with a block size of 128 bits (and S-boxes that take input of size 8). Does the block size make any difference?

5.2 Our attack on a three-round substitution-permutation network does not recover the key but only shows how to distinguish it from a random permutation. Thus it is not a "complete break". Despite this, show that using a three-round substitution-permutation network with *counter mode* (see Section 3.6.4) can have disastrous effects on the security of encryption.

5.3 Consider a modified substitution-permutation network where instead of carrying out key-mixing, substitution and permutation in alternating order, first R keys are mixed (via XOR), then R substitution steps are carried out, and finally R permutations are run. Analyze the security of this construction. Does it make a difference if the same S-boxes and permutations are used in all steps or if they are different?

5.4 Show that in the DES structure, $DES_k(x) = \overline{DES_{\overline{k}}(\overline{x})}$ for every key k and input x (where $\overline{z}$ denotes the bitwise complement of z). This is called the *complementary property* of DES.

5.5 Use the previous exercise to show how it is possible to find the DES secret key in time 2^{55} (i.e., half the time of a straightforward brute-force search). For your attack, you can assume that you are given an oracle that computes DES under the secret key k.

5.6 Provide a *formal proof* that the initial and final permutations IP and IP^{-1} have no effect on the security of DES (beyond slowing down attacks by a constant factor).

> **Hint:** Show that any attack on DES without the permutations can be converted into an attack on DES with the permutations.

5.7 This exercise discusses problematic keys for DES; these keys are called *weak keys*:

(a) Assume that the DES secret key k equals 0^{56}. Show that for every x 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. Present the keys and explain why they have the property.

(c) Is the existence of these 4 keys a threat to the security of DES? Explain your answer.

5.8 Describe an algorithm that is given two sorted lists of length N and finds all values that appear in both lists in time $\mathcal{O}(N)$. Recall that this is used in the attack on double-DES.

5.9 It has been proposed to use double-DES as a fixed-length collision-resistant hash function within the Merkle-Damgård transform, in the following way. Define the hash function $h : \{0,1\}^{112} \to \{0,1\}^{64}$ as $h(x_1 \| x_2) = DES_{x_1}(DES_{x_2}(0^{64}))$ where $|x_1| = |x_2| = 56$.

(a) Write down an actual collision in this fixed-length hash function.

(b) In how much time is it possible to find a *preimage* for the hash function. That is, given some $y \in \{0,1\}^{64}$ show how to find a pair (x_1, x_2) such that $h(x_1 \| x_2) = y$. Make your attack as quick as possible. Does your attack make it possible to find a preimage that "makes sense" (say, one that contains an English word)?

(c) What happens when h as above is used in the Merkle-Damgård transform? Find the most effective attack that you can think of.

5.10 Describe a detailed attack on all of the following (silly) modifications to DES:

(a) The S-boxes all output zeroes, irrespective of the input

(b) The internal f function computes the identity function

(c) Instead of using different subkeys in every round, the same 48-bit key is used in every round of the Feistel structure.

If you claim insecurity, then describe a detailed attack and analyze its complexity.

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 and pseudorandom functions. In addition, we showed that these objects are the basic building blocks for constructing secure encryption schemes (in Chapter 4 we also saw that they can be used for constructing message authentication codes). Finally, in Chapter 5 we studied how pseudorandom functions can be constructed *heuristically*. Thus, all of our constructions of encryption schemes and message authentication codes can be proven secure under the *assumption* that pseudorandom generators and functions exist (and thus under the assumption that constructions like the AES block cipher indeed constitute pseudorandom functions). Since these objects form the basis of essentially all of "private-key cryptography", it is of great importance to enhance our understanding of them from a theoretical (and more rigorous) point of view. In this chapter we study pseudorandom generators and functions and show under what (minimal) assumptions they can be constructed, and how.

That material in this chapter is for the most part theoretical, and we do not suggest that the constructions presented here should (or could) be used in practice. Indeed, they are far too inefficient for that. Nevertheless, a strong theoretical understanding of pseudorandomness greatly deepens our understanding of how security is achieved, and what assumptions are necessary. In addition, a strong theoretical understanding is often beneficial when analyzing schemes used in practice.

A note regarding this chapter. This chapter is somewhat more advanced and more theoretical than the others in the book. The chapter may be skipped entirely and is not relied on in the rest of the book. (The one exception to this rule is that we do mention one-way functions later on. Nevertheless, if desired, they can be taken at the intuitive level.) Having said this, we have made great efforts to make the material here suitable for an advanced undergraduate or beginning graduate audience. This is especially true for Sections 6.1 and 6.2 that we believe are suitable for a general undergraduate audience. 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

As we have mentioned, the basic aim of this chapter is to understand the *minimal assumption* required for constructing pseudorandom generators and functions, and how this assumption is used in those constructions. But, why is any assumption necessary? Why can't we construct a pseudorandom function from scratch, and just prove mathematically that it is indeed pseudorandom? The answer to this question is simply that an unconditional proof of the existence of pseudorandom generators or functions would involve breakthroughs in complexity theory that seem far beyond reach today.

Given that some assumption is necessary, at least with our current understanding of complexity, a natural goal is to try to base our constructions on the "minimal assumption" possible. Formally, a minimal assumption is one that is both *necessary* and *sufficient* for achieving constructions of pseudorandom generators and functions. As it turns out, the minimal assumption here is that of the existence of *one-way functions*. Loosely speaking, such a function has the property that it is easy to compute, but (almost always) hard to invert. The fact that this assumption is indeed minimal will be proven throughout this chapter, and in particular in Section 6.7.

We remark that one-way functions are in fact a minimal assumption for almost all of cryptography, and not just for obtaining pseudorandom generators and functions. (The only exception are cryptographic tasks for which no computational assumptions are needed.) Furthermore, on an *intuitive level* they constitute a far weaker assumption than the existence of pseudorandom generators or functions (although technically speaking, as we show in this chapter they are actually equivalent and either they both exist or they both do not exist).

6.1.1 Definitions

One-way functions have the property that they are easy to compute, but hard to invert. Since we are interested in a computational task that is almost always hard to solve, the hard-to-invert requirement is formalized by saying that a polynomial-time adversary will fail to invert the function (i.e., find *some* preimage), except with negligible probability. (Note that it is always possible to succeed with negligible probability, by just guessing a preimage of the appropriate length. Likewise, given exponential-time, it is always possible to search the entire domain for a preimage.)

Simplifying convention. Throughout this entire chapter, we will assume that the input and output lengths of every one-way function (or variant) f are *polynomially related*. This means that there exist two constants c_1 and c_2 such that for every x, it holds that $|x|^{1/c_1} \leq |f(x)| \leq |x|^{c_2}$. We remark that the requirement that $f(x)$ be at most of length $|x|^{c_2}$ is given in any case by

the fact that f must be efficiently computable (a polynomial-time algorithm cannot write more than polynomially many bits). In contrast, the requirement that $f(x)$ be at least of length $|x|^{1/c_1}$ is a simplifying convention. We remark that there are ways of dealing with one-way functions for which this doesn't hold. However, this convention simplifies the notation and so we adopt it.

DEFINITION 6.1 (one-way functions): *A function $f : \{0,1\}^* \to \{0,1\}^*$ is called* one-way *if the following two conditions hold:*

1. Easy to compute: *There exists a polynomial-time algorithm M_f such that on input any $x \in \{0,1\}^*$, M_f outputs $f(x)$ (i.e., $M_f(x) = f(x)$ for every x).*

2. Hard to invert: *For every probabilistic polynomial-time inverting algorithm $\mathcal{A}$, there exists a negligible function* negl *such that*

$$\Pr\left[\mathcal{A}(f(x)) \in f^{-1}(f(x))\right] \leq \mathsf{negl}(n) \tag{6.1}$$

   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 it is only guaranteed that a one-way function is hard to invert when the input is *uniformly distributed*. Thus, there may be many inputs (albeit a negligible fraction) for which the function can be inverted. Furthermore, as usual for asymptotic definitions, it is only guaranteed that it be hard to invert for all long enough values of x.

Successful inversion of one-way functions. A very important point to note also that a function that is *not* one-way is not necessarily easy to invert all the time (or even "often"). Rather, the converse 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 $f(x)$ for $x \in \{0,1\}^n$ with probability at least $\varepsilon(n)$. What this actually means is that there exists a positive polynomial $q(\cdot)$ such that for *infinitely many n's*, 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, then the function is not one-way. This holds even though $\mathcal{A}$ only succeeds on half of the values of n and even though when it succeeds, it is only with probability n^{-10}. We also stress that the inverting algorithm is not required to find the exact x used in computing $y = f(x)$. Rather, if it finds any value x' such that $f(x') = y = f(x)$, then it has succeeded in its task.

Exponential-time inversion. As we have mentioned, any one-way function can be inverted given enough time. Specifically, given a value y, it is always possible to simply try all values x of increasing length (up to some bound) 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. However, it cannot be solved efficiently.

One-way permutations. We will often be interested in one-way functions with special properties. One particular category of interest is that of one-way functions that are bijections. We call such functions "one-way permutations". Since we are considering infinite domains (i.e., functions that receive inputs of all lengths), we should explain in more detail what we mean. We will call a function over an infinite domain a permutation if for every n, the function *restricted to inputs of length* n is a bijection.

DEFINITION 6.2 (one-way permutations): *Let f be a function with domain $\{0,1\}^*$ and define the function f_n to be the restriction of f to the domain $\{0,1\}^n$ (i.e., for every $x \in \{0,1\}^n$, $f_n(x) = f(x)$). Then, a one-way function f is called a* one-way permutation *if for every n, the function f_n is 1–1 and onto $\{0,1\}^n$.*

An interesting property of one-way permutations is that any value y uniquely determines its preimage x. This is due to the fact that a permutation f is a bijection, and so there exists only one preimage. Thus, even though y fully determines x, it is still hard to find x in polynomial-time.

We remark that a more involved notion called "families of one-way permutations" is typically considered in the cryptography literature. Nevertheless, for the sake of clarity, we will consider the above simpler notion and remark that it makes almost no difference to the material presented in this chapter.

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 candidates that we have for one-way functions and permutations actually don't fit into this type of formalization. Rather, for every n there is a *different* function with a finite domain and range containing inputs of size $\mathsf{poly}(n)$. Furthermore, the inputs and outputs may not be mere strings of size n but may have a certain form. For example, for every prime p we may consider a permutation f_p over $\mathbb{Z}_p$. The collection of all f_p then constitutes a *family of functions*. This brings us to the following definition:

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 we define next.*

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.*

Consider a function f_p that is defined for every prime p as follows: $f_p(x) = g^x \bmod p$. Then, one can define a family of functions where Gen chooses a random prime of the appropriate length n (in this case $I = p$), $\mathsf{Samp}(I)$ chooses a random element x within $\mathbb{Z}_p$, and $f(x)$ computes $g^x \bmod p$. It is not hard to see that this is actually a family of permutations. We now proceed to define one-wayness for a family of functions or permutations. We begin by defining an "inversion" experiment:

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 is one-way if success in the above experiment occurs with at most negligible probability. That is:

DEFINITION 6.4 *A family of functions/permutations $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ is called* one-way *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Invert}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

6.1.2 Candidates

One-way functions are only of interest if they actually exist. Since we do not know how to prove that they exist (because this would imply a major breakthrough in complexity theory), we conjecture or assume their existence. This conjecture (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*. This problem relates to the difficulty of finding the prime factors of a number that is the product of long uniformly distributed primes of similar length. This leads us to define the function $f_{\text{mult}}(x \| y) = x \cdot y$. Now, if there is no restriction on the lengths of x and y, then f_{mult} is easy to invert (with

high probability $x \cdot y$ will have a small prime factor p that can be found, and then it is possible to return $(p, xy/p)$ as a preimage). Nevertheless, there are two ways to modify f_{mult} so that it yields a one-way function. The first is to require that $|x| = |y|$ (this prevents finding a small prime factor), and the other is to use the input to sample two primes of approximately the same size (see Section 7.2.1). The integer factorization problem is discussed in greater length in Chapters 7 and 8.

Another candidate one-way function 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 all x_i's are of length n, and J is a subset of $\{1, \ldots, n\}$. Note that when given an image $(x_1, \ldots, x_n, y)$ of this function, the task of inverting it is exactly that of finding a subset J' of $\{1, \ldots, n\}$ such that $\sum_{j \in J} x_j = y$.[1]

We conclude with a family of permutations that is widely believed to be one-way. This family is based on the so-called *discrete logarithm problem* and is defined by $f_p(x) = g^x \bmod p$ for any prime p. We described this family above and remark here that it is believed to be one-way. This is called the discrete logarithm because the logarithm function is the inverse of exponentiation; it is "discrete" because we work over $\mathbb{Z}_p$ and not the reals.

In summary, one-way functions and one-way permutations are assumed to exist, and we have a number of concrete candidates for their existence. We will study some of these in detail in Chapters 7, 8 and 11.

6.1.3 Hard-Core Predicates

By the definition, a one-way function is hard to invert. Stated differently, given a value $y = f(x)$, the value of x is unknown to any polynomial-time inverting algorithm. Thus, f hides information about x, even when $f(x)$ is known. Recall that when f is a permutation, $f(x)$ fully determines x. Nevertheless, x is still hidden to any polynomial-time algorithm.

One may have the impression that this means that x is completely unknown, even given $f(x)$. However, this is *not* the case. Indeed, a one-way function f may yield a lot of information about its input, and yet still be hard to invert. For example, let f be a one-way function and define $g(x_1, x_2) = (f(x_1), x_2)$, where $|x_1| = |x_2|$. Then, it is easy to show that g is also a one-way function, even though it reveals half of its input (the proof that g is one-way is left as an exercise). For our applications, we will need to classify what information is truly hidden by f. This is exactly the purpose of a hard-core predicate.

[1] We remark that students who have taken a course in complexity or who have studied $\mathcal{NP}$-completeness may be familiar with the subset-sum problem and the fact that it is $\mathcal{NP}$-complete. We stress that $\mathcal{NP}$-completeness does not imply one-wayness because $\mathcal{NP}$-completeness relates to worst-case complexity and not average case as we consider here. Thus, our belief that this function is one-way is based on the lack of known algorithms to solve this problem, and not on the fact that the general problem is $\mathcal{NP}$-complete.

Loosely speaking, a hard-core predicate hc of a function f is a function outputting a single bit with the following property: If f is one-way, then upon input $f(x)$ it is infeasible to correctly guess $\mathsf{hc}(x)$ with any non-negligible advantage above $1/2$. (Note that it is always possible to compute $\mathsf{hc}(x)$ correctly with probability $1/2$ by just randomly guessing it.)

Notation: Before proceeding to the definition, we remark that in this chapter we will often make the probability space explicit by subscripting it in the probability (see the equation in the definition below). We found this to be clearer for the material in this chapter.

We now proceed with the definition of a hard-core predicate.

DEFINITION 6.5 (hard-core predicate): *A polynomial-time computable predicate* $\mathsf{hc} : \{0,1\}^* \to \{0,1\}$ *is called a* hard-core of a function f *if 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* $\mathcal{A}$*.*

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 preimage. 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. (Actually, it can be shown that for every given predicate hc, there exists a one-way function for which hc is *not* a hard-core of f.)

Meaningless hard-core predicates. We note that some functions have "meaningless" hard-core predicates. For example, let f be any function and define $g(\sigma, x) = f(x)$ where $\sigma \in \{0,1\}$ and $x \in \{0,1\}^n$. Then, given $g(\sigma, x)$ it is clearly hard to guess σ with probability greater than $1/2$ (because σ is not determined by $g(\sigma, x)$). This holds even if f (and thus g) is not one-way.

In contrast, a 1–1 function f has a hard-core predicate only if it is one-way (see Exercise 6.9). Intuitively, this is the case because when a function is 1–1, the value $f(x)$ fully determines x. Now, if f is *not* one-way, then it can be inverted, revealing the unique preimage x. Thus, $\mathsf{hc}(x)$ can be computed from $f(x)$ with some non-negligible advantage. This shows that if

a 1–1 function is not one-way, then it cannot have a hard-core predicate. We stress that in the example above of $g(\sigma, x)$, the fact that σ remains hidden is due to the fact that it is not determined. In contrast, for 1–1 functions, the difficulty of guessing the hard-core is due to *computational difficulty.* We will use hard-core predicates in order to construct pseudorandom generators. In that construction, it will become clear why a hard-core predicate that hides an undetermined bit is of no use.

6.2 Overview of Constructions

In this section, we describe the steps in the construction of pseudorandom generators and functions from one-way functions. The first step is to show that for every one-way function f, there exist a hard-core predicate for the function $g(x, r) = (f(x), r)$, where $|x| = |r|$ (recall that if f is one-way then so is g). Notice that if f was 1–1 or a permutation, then so is g. Thus, nothing is lost by considering g instead of f. We have the following theorem:

THEOREM 6.6 *Let f be a one-way function and define $g(x, r) = (f(x), r)$. Then, the function $\mathsf{gl}(x, r) = \bigoplus_{i=1}^{n} x_i \cdot r_i$, where $x = x_1, \ldots, x_n$ and $r = r_1, \ldots, r_n$, is a hard-core predicate of g.*

Notice that the function $\mathsf{gl}(x, r)$ outputs a bit that consists of the exclusive-or of a *random subset* of the bits $x_1, \ldots, x_n$. This is due to the fact that r can be seen as selecting this subset (when $r_i = 1$ the bit x_i is included in the XOR, and otherwise it is zeroed), and r is uniformly distributed. Thus, Theorem 6.6 essentially states that $f(x)$ hides the exclusive-or of a random subset of the bits of x. ¿From here on, we will write $f(x)$ as the one-way function, and unless explicitly stated otherwise, it should be understood that we really mean a one-way function of the form g as above. We remark that the function is denoted gl after Goldreich and Levin who proved Theorem 6.6.

The next step in the construction is to show how hard-core predicates can be used to obtain pseudorandom generators. Despite the fact that more general theorems exist, we will show this only for the case of one-way permutations. We use the following two important facts. First, if f is a permutation, then for a uniformly distributed x it holds that $f(x)$ is also uniformly distributed. Second, if hc is a hard-core of f, then the bit $\mathsf{hc}(x)$ looks random, even given $f(x)$. (This second fact follows from the fact that a polynomial-time algorithm can guess the value $\mathsf{hc}(x)$ given $f(x)$ with probability only negligibly greater than $1/2$. This is equivalent to say that it looks random, or more formally, that it is pseudorandom.) We have the following theorem:

THEOREM 6.7 *Let f be a one-way permutation and let hc be a hard-core predicate of f. Then, $G(s) = (f(s), \mathsf{hc}(s))$ constitutes a pseudorandom generator with expansion factor $\ell(n) = n + 1$.*

The existence of a pseudorandom generator that stretches the seed even by just a single bit is very non-trivial. The possibility of creating even a single bit of randomness in a deterministic way is very surprising (note, for example, that such a result cannot be achieved in a world where algorithms are not computationally bounded). Despite this, we are interested in obtaining *many* pseudorandom bits. This is due to the fact that in order to encrypt with pseudorandom generators (as in Section 3.4) it is necessary to have pseudorandom generators with large expansion factors. Fortunately, pseudorandom generators that stretch the seed by one bit can be used to construct pseudorandom generators with any polynomial expansion factor.

THEOREM 6.8 *Assume that there exist pseudorandom generators with expansion factor $\ell(n) = n + 1$. Then, for every polynomial $p(\cdot)$, there exists a pseudorandom generator with expansion factor $\ell(n) = p(n)$.*

Thus, pseudorandom generators can be constructed from any one-way permutation. Pseudorandom generators suffice for obtaining secure encryption in the presence of eavesdropping adversaries. However, for active CPA and CCA attacks and for constructing message authentication code, we relied upon pseudorandom functions. The last step is thus the construction of pseudorandom functions from pseudorandom generators. That is,

THEOREM 6.9 *Assume that there exist pseudorandom generators with expansion factor $\ell(n) = 2n$. Then, there exist pseudorandom functions and pseudorandom permutations.*

Combining Theorems 6.6 to 6.9, we have the following corollary:

COROLLARY 6.10 *Assuming the existence of one-way permutations, there exist pseudorandom generators with any polynomial expansion factor, pseudorandom functions and pseudorandom permutations.*

We remark that it is actually possible to obtain all of these results from *any* one-way function (without requiring that it be a permutation or even 1–1). However, the construction that is based on arbitrary one-way functions is very involved and is out of the scope of this book.

6.3 Hard-Core Predicates from Every One-Way Function

In this section, we prove Theorem 6.6. We begin by restating the theorem:

THEOREM 6.11 (hard-core predicate – restated): *Let f be a one-way function and let g be defined by $g(x,r) = ((f(x),r))$, where $|x| = |r|$. Let $\mathsf{gl}(x,r) = \bigoplus_{i=1}^{n} x_i \cdot r_i$ be the inner product function, where $x = x_1 \cdots x_n$ and $r = r_1 \cdots r_n$. Then, the predicate gl is a hard-core of the function g.*

We now proceed to prove Theorem 6.11. Due to the complexity of the proof, we present it in three stages beginning with the most simplistic case and ending with the full proof.

6.3.1 The Most Simplistic Case

We first show that if there exists a polynomial-time adversary $\mathcal{A}$ that always successfully guesses $\mathsf{gl}(x,r)$ given $(f(x),r)$, 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 6.12 *Let f and gl be as in Theorem 6.11. If there exists a probabilistic polynomial-time adversary $\mathcal{A}$ such that for infinitely many n's*

$$\Pr_{x,r \leftarrow \{0,1\}^n}\left[\mathcal{A}((f(x),r)) = \mathsf{gl}(x,r)\right] = 1$$

then there exists a probabilistic polynomial-time adversary $\mathcal{A}'$ such that for infinitely many n's

$$\Pr_{x \leftarrow \{0,1\}^n}\left[\mathcal{A}'(f(x)) \in f^{-1}(f(x))\right] = 1$$

PROOF Let $\mathcal{A}$ be as in the proposition and consider the infinite set of n's upon which $\mathcal{A}$ successfully guesses $\mathsf{gl}(x,r)$. We construct an adversary $\mathcal{A}'$ as follows. Let e^i denote the length-n vector that contains zeroes in all positions, except for the i^{th} position where it contains a one. Then, upon input y, adversary $\mathcal{A}'$ invokes $\mathcal{A}$ with input (y, e^i) for $i = 1, \ldots, n$. That is, for $y = f(x)$, $\mathcal{A}'$ effectively invokes $\mathcal{A}$ with input $g(x, e^i)$. By the assumption in the proposition (that $\mathcal{A}$ always succeeds in guessing gl), we have that for

each i, $\mathcal{A}$ outputs $\mathsf{gl}(x, e^i)$. However,

$$\mathsf{gl}(x, e^i) = \bigoplus_{j=1}^{n} x_j \cdot e^i_j = x_i \cdot e^i_i + \bigoplus_{j \neq i} x_j \cdot e^i_j = x_i$$

where the last equality is due to the fact that $e^i_i = 1$, but for all $j \neq i$ it holds that $e^i_j = 0$. We therefore conclude that $\mathcal{A}'$ obtains the bits $x_1, \ldots, x_n$ one at a time from $\mathcal{A}$. Once it concludes, it outputs $x = x_1, \ldots, x_n$. By the assumption on $\mathcal{A}$, we have that $f(x) = y$, and so $\mathcal{A}'$ successfully inverts f. ■

As we have mentioned, by the assumption that f is one-way, it is impossible to invert f with non-negligible probability (let alone probability 1) in polynomial-time. Thus, we conclude that an adversary $\mathcal{A}$ that can guess $\mathsf{gl}(x, r)$ with probability 1 does not exist. Of course, this is still very far from our ultimate goal of showing that $\mathsf{gl}(x, r)$ can be guessed with probability only negligibly greater than 1/2.

6.3.2 A More Involved Case

We now proceed to show that it is hard to guess $\mathsf{gl}(x, r)$ with probability that is non-negligibly greater than 3/4. This already establishes a measure of unpredictability of $\mathsf{gl}(x, r)$ given $f(x, r)$. Notice that the strategy in the proof of Proposition 6.12 fails completely here because it may be that $\mathcal{A}$ never succeeds when it receives $r = e^i$. Furthermore, notice that in this case, $\mathcal{A}'$ cannot know if a particular bit output by $\mathcal{A}$ as a guess for $\mathsf{gl}(x, r)$ is correct or not. The only fact that $\mathcal{A}'$ can deduce is that with probability non-negligibly greater than 3/4, adversary $\mathcal{A}$ is indeed correct. We now prove the following:

PROPOSITION 6.13 *Let f and gl be as in Theorem 6.11. If there exists a probabilistic polynomial-time adversary $\mathcal{A}$ and a polynomial $p(\cdot)$ such that for infinitely many n's it holds*

$$\Pr_{x,r \leftarrow \{0,1\}^n}\left[\mathcal{A}(f(x), r) = \mathsf{gl}(x, r)\right] \geq \frac{3}{4} + \frac{1}{p(n)}$$

then there exists a probabilistic polynomial-time adversary $\mathcal{A}'$ and a polynomial $p'(\cdot)$ such that for infinitely many n's

$$\Pr_{x \leftarrow \{0,1\}^n}\left[\mathcal{A}'(f(x)) \in f^{-1}(f(x))\right] \geq \frac{1}{p'(n)}$$

PROOF The main observation behind 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 i^{th} bit of x. This follows from the following calculation:

$$\begin{aligned}&\mathsf{gl}(x,r) \oplus \mathsf{gl}(x, r \oplus e^i) \\ &= \left(\bigoplus_{j=1}^{n} x_j \cdot r_j\right) \oplus \left(\bigoplus_{j=1}^{n} x_j \cdot (r_j \oplus e^i)\right) = x_i \cdot r_i \oplus x_i \cdot (r_i \oplus 1) = x_i\end{aligned}$$

where the second equality is due to the fact that for all $j \neq i$, the value $x_j \cdot r_j \oplus x_j \cdot r_j$ appears (and so is cancelled out).

In order to use this observation, we will need to choose many different values of r for a fixed x (the reason for this will become apparent later). We therefore need to show that for many x's, the probability that $\mathcal{A}$'s correctly outputs both $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x, r \oplus e^i)$, when r is chosen uniformly, is good. In the following claim, we show that there is a large enough set of concrete "good inputs" x upon which $\mathcal{A}$ often succeeds. This claim allows us to consider the probability distribution over the choice of r (and not over the uniform choice of both x and r), which makes things easier.

CLAIM 6.14 *For infinitely many n's, 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*

$$\Pr_{r \leftarrow \{0,1\}^n}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r)] \geq \frac{3}{4} + \frac{1}{2p(n)} \tag{6.2}$$

where the probability is taken over the choice of r only.

PROOF Set $\varepsilon(n) = 1/p(n)$ and for any given x, let

$$s(x) = \Pr_{r \leftarrow \{0,1\}^n}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r)].$$

Then, 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). We show that $|S_n| \geq \frac{\varepsilon(n)}{2} \cdot 2^n$. This follows from a simple averaging argument. (That is, if $\mathcal{A}$ inverts with probability $3/4 + \varepsilon(n)$, then there must be at least an $\varepsilon(n)/2$ fraction of inputs for which it succeeds with probability $3/4 + \varepsilon(n)/2$.) We have:

$$\begin{aligned}&\Pr_{x,r}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r)] \\ &\quad = \Pr_{x,r}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r) \mid x \in S_n] \cdot \Pr_{x}[x \in S_n] \\ &\qquad + \Pr_{x,r}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r) \mid x \notin S_n] \cdot \Pr_{x}[x \notin S_n] \\ &\quad \leq \Pr_{x}[x \in S_n] + \Pr_{x,r}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r) \mid x \notin S_n]\end{aligned}$$

and so

$$\begin{aligned}&\Pr_{x}[x \in S_n] \\ &\quad \geq \Pr_{x,r}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r)] - \Pr_{x,r}[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r) \mid x \notin S_n]\end{aligned}$$

By the definition of S_n, it holds that 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 we have that

$$\Pr_x[x \in S_n] \geq \frac{3}{4} + \varepsilon(n) - \frac{3}{4} - \frac{\varepsilon(n)}{2} = \frac{\varepsilon(n)}{2}$$

This implies that S_n must be at least of size $\frac{\varepsilon(n)}{2} \cdot 2^n$ (because x is uniformly distributed in $\{0,1\}^n$), completing the proof of the claim. ■

For the rest of the proof we set $\varepsilon(n) = 1/p(n)$ and consider only the infinite set of n's upon which $\mathcal{A}$ succeeds with probability $\varepsilon(n)$. The claim above states that for an $\varepsilon(n)/2 = 1/2p(n)$ fraction of inputs x, the adversary $\mathcal{A}$ correctly outputs $\mathsf{gl}(x, r)$ with probability at least $3/4 + \varepsilon(n)/2$, when r is uniformly distributed. Below, we will show that $\mathcal{A}'$ can invert $f(x)$ with good probability in the case that x is as above (i.e., for $x \in S_n$). Since $\varepsilon(\cdot)$ is a non-negligible function, this is enough to contradict the one-wayness of f (because for a one-way function it must be possible to invert only a *negligible fraction* of the inputs with good probability). Formally, it suffices to look only at $x \in S_n$ because

$$\Pr[\mathcal{A}'(f(x)) = x] \geq \Pr[x \in S_n] \cdot Pr[\mathcal{A}'(f(x)) = x \mid x \in S_n].$$

From now on, we focus only on $x \in S_n$.

Now, let $\mathcal{A}$ be as in the proposition. Then, given $y = f(x)$ for any $x \in S_n$, an adversary $\mathcal{A}'$ can choose a random r and invoke $\mathcal{A}$ on (y, r). By the assumption in the proposition, it holds that $\mathcal{A}$ outputs $\mathsf{gl}(x, r) = \bigoplus_{j=1}^{n} x_j \cdot r_j$ with probability at least $3/4 + \varepsilon(n)/2$. Stated otherwise, $\mathcal{A}$ *fails* to output the correct $\mathsf{gl}(x, r)$ with probability *at most* $1/4 - \varepsilon(n)/2$. Likewise, upon input $(y, r \oplus e^i)$, we have that $\mathcal{A}$ outputs $\mathsf{gl}(x, r \oplus e^i)$ with probability at least $3/4 + \varepsilon(n)/2$ and so fails with probability at most $1/4 - \varepsilon(n)/2$. This holds because if r is uniformly distributed then so is $r \oplus e^i$.

Recall that if $\mathcal{A}$ outputs the correct values for both $\mathsf{gl}(x, r)$ and $\mathsf{gl}(x, r \oplus e^i)$ then $\mathcal{A}'$ will obtain the correct value x_i. We would now like to analyze the probability that $\mathcal{A}$ outputs the correct values for *both* $\mathsf{gl}(x, r)$ and $\mathsf{gl}(x, r \oplus e^i)$. Note that r and $r \oplus e^i$ are not independent. Therefore, we cannot just multiply the probabilities of success; that is, we cannot claim that the probability is $(3/4 + \varepsilon(n)/2) \cdot (3/4 + \varepsilon(n)/2) > 9/16$. Nevertheless, we can use the union bound (see Proposition A.7 in Appendix A) and just sum the probabilities of failure. Thus, we have that the probability that $\mathcal{A}$ is *incorrect* on at least one of $\mathsf{gl}(x, r)$ 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 means that the adversary $\mathcal{A}'$, upon input $f(x)$, is able to

correctly guess the i^{th} bit of x with probability that is non-negligibly greater than $1/2$ (when $x \in S_n$). This does not yet suffice because in order to invert $f(x)$, $\mathcal{A}$ must simultaneously guess all of the bits of the preimage $x_1, \ldots, x_n$ correctly. However, this can be achieved by separately running the above procedure for each bit, while reducing the error probability for each guess. In order to see how this works, assume that we can improve the above procedure so that $\mathcal{A}'$ obtains a correct guess of x_i with probability at least $1 - \frac{1}{2n}$. Since this holds for each i, we have the following: $\mathcal{A}'$ incorrectly guesses a bit x_i with probability at most $\frac{1}{2n}$. Therefore, by the union bound, the probability that $\mathcal{A}'$ will guess at least one of $x_1, \ldots, x_n$ incorrectly is at most $n \cdot \frac{1}{2n} = \frac{1}{2}$. Stated differently, $\mathcal{A}'$ obtains the correct preimage $x = x_1 \cdots x_n$ with probability at least $1/2$ and so successfully inverts $f(x)$, in contradiction to the one-wayness of f.

It remains to show how we can improve the procedure of $\mathcal{A}'$ for guessing x_i so that it is correct with probability at least $1 - \frac{1}{2n}$. The idea behind the improved procedure is to run the original procedure many times (using independent coins each time). Then, since the correct answer is obtained more often than the incorrect one (with an advantage of $\varepsilon(n)$), it holds that over many trials – proportionate to $n/\varepsilon(n)$ – the majority result will be correct with high probability (if a coin that is biased towards heads is flipped many times, then with very high probability heads will appear more often than tails). This can be proven using the Chernoff bound (a standard bound from probability theory). We remark that the Chernoff bound only works for independent random variables. However, since $x \in S_n$ is fixed, the probability is only over the choice of r, which is chosen independently in each attempt. Therefore, each trial is independent.[2] We leave the application of the Chernoff bound here (so as to complete the proof) as an exercise. ■

A corollary of Proposition 6.13 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

We remark that this section is more advanced than the rest of the book, and relies on more involved concepts from probability theory and theoretical computer science (for example, the proof relies heavily on the notion of pairwise

[2] This is the reason that we needed to fix the set S_n. Otherwise, we would have a random x and a random r, and in the different trials we would only be changing r. Such trials would not be independent, and so Chernoff could not be used. We note that the random coins of $\mathcal{A}$ can be chosen independently each time and so pose no problem.

independent random variables). We include the full proof for completeness, and for more advanced students and courses.

6.3.3.1 Preliminaries – Markov and Chebyshev Inequalities

Before proceeding to the full proof of Theorem 6.11, we prove two important inequalities that we will use. These inequalities are used to measure the probability that a random variable will significantly deviate from its expectation.

Markov Inequality: *Let X be a non-negative random variable and v a real number. Then:*

$$\Pr[X \geq v \cdot \mathsf{Exp}[X]] \leq \frac{1}{v}$$

Equivalently: $\Pr[X \geq v] \leq \mathsf{Exp}[X]/v$.

PROOF

$$\begin{aligned}\mathsf{Exp}[X] &= \sum_x \Pr[X = x] \cdot x \\ &\geq \sum_{x<v} \Pr[X = x] \cdot 0 + \sum_{x \geq v} \Pr[X = x] \cdot v \\ &= \Pr[X \geq v] \cdot v\end{aligned}$$

■

The Markov inequality is extremely simple, and is useful when very little information about X is given. However, when an upper-bound on its variance is known, better bounds exist. Recall that $\mathsf{Var}(X) \stackrel{\text{def}}{=} \mathsf{Exp}[(X - \mathsf{Exp}[X])^2]$, that $\mathsf{Var}(X) = \mathsf{Exp}[X^2] - \mathsf{Exp}[X]^2$, and that $\mathsf{Var}[aX + b] = a^2\mathsf{Var}[X]$.

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 We define a random variable $Y \stackrel{\text{def}}{=} (X - \mathsf{Exp}[X])^2$ and then apply the Markov inequality to $\Pr[Y \geq \delta^2]$. That is,

$$\begin{aligned}\Pr[|X - \mathsf{Exp}[X]| \geq \delta] &= \Pr[(X - \mathsf{Exp}[X])^2 \geq \delta^2] \\ &\leq \frac{\mathsf{Exp}[(X - \mathsf{Exp}[X])^2]}{\delta^2} \\ &= \frac{\mathsf{Var}(X)}{\delta^2}\end{aligned}$$

where the second inequality is by the "equivalent" formulation of Markov. ■

An important corollary of Chebyshev's inequality relates to pairwise independent random variables. A series of random variables $X_1, \ldots, X_m$ are called pairwise independent if for every $i \neq j$ and every a and b it holds that

$$\Pr[X_i = a \ \wedge \ X_j = b] = \Pr[X_i = a] \cdot \Pr[X_j = b]$$

We note that for pairwise independent random variables $X_1, \ldots, X_m$ it holds that $\mathsf{Var}[\sum_{i=1}^m X_i] = \sum_{i=1}^m \mathsf{Var}[X_i]$ (this is due to the fact that every pair of variables are independent and so their covariance equals 0). (Recall that $\mathsf{cov}(X,Y) = \mathsf{Exp}[XY] - \mathsf{Exp}[X]\mathsf{Exp}[Y]$ and $\mathsf{Var}[X+Y] = \mathsf{Var}[X] + \mathsf{Var}[Y] - 2\mathsf{cov}(X,Y)$. This can be extended to any number of random variables.)

COROLLARY 6.15 (pairwise-independent sampling): *Let $X_1, \ldots, X_m$ be pairwise-independent random variables with the same expectation μ and the same variance σ^2. Then, 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 the linearity of expectations, $\mathsf{Exp}[\sum_{i=1}^m X_i/m] = \mu$. Applying Chebyshev's inequality, we have

$$\Pr\left[\left|\frac{\sum_{i=1}^m X_i}{m} - \mu\right| \geq \varepsilon\right] \leq \frac{\mathsf{Var}\left(\sum_{i=1}^m \frac{X_i}{m}\right)}{\varepsilon^2}$$

By pairwise independence, it follows that

$$\mathsf{Var}\left(\sum_{i=1}^m \frac{X_i}{m}\right) = \sum_{i=1}^m \mathsf{Var}\left(\frac{X_i}{m}\right) = \frac{1}{m^2}\sum_{i=1}^m \mathsf{Var}(X_i) = \frac{1}{m^2}\sum_{i=1}^m \sigma^2 = \frac{\sigma^2}{m}$$

The inequality is obtained by combining the above two equations. ■

6.3.3.2 The Full Proof of the Hard-Core Predicate

Similarly to above, we prove Theorem 6.11 via the following proposition:

PROPOSITION 6.16 *Let f and gl be as in Theorem 6.11. If there exists a probabilistic polynomial-time adversary $\mathcal{A}$ and a polynomial $p(\cdot)$ such that for infinitely many n's it holds that*

$$\Pr_{x,r \leftarrow \{0,1\}^n}\left[\mathcal{A}(f(x), r) = \mathsf{gl}(x,r)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

then there exists a probabilistic polynomial-time adversary $\mathcal{A}'$ and a polynomial $p'(\cdot)$ such that for infinitely many n's

$$\Pr_{x\leftarrow\{0,1\}^n}\left[\mathcal{A}'(f(x))\in f^{-1}(f(x))\right]\geq\frac{1}{p'(n)}$$

PROOF Before we prove the proposition, we remark that it implies Theorem 6.11 because it states that if it is possible to guess the hard-core predicate with any non-negligible advantage, then it is possible to invert f with non-negligible advantage. Thus, the assumed one-wayness of f implies that gl is a hard-core predicate of f.

As in the proof of Proposition 6.13, we set $\varepsilon(n)=1/p(n)$ and focus only on the infinite set of n's upon which $\mathcal{A}$ succeeds with probability $\varepsilon(n)$. We also begin by defining a set S_n of inputs $x\in\{0,1\}^n$ for which $\mathcal{A}$ is successful in guessing $\mathsf{gl}(x,r)$, when r is randomly chosen. The following claim is analogous to the claim presented in the case where $\mathcal{A}$ is assumed to succeed with probability greater than $3/4$.

CLAIM 6.17 *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*

$$s(x)=\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)]\geq\frac{1}{2}+\frac{\varepsilon(n)}{2}$$

Claim 6.17 is proved in an almost identical way to Claim 6.14 and we therefore leave it as an exercise.

In the proof of Proposition 6.13, we showed how to convert $\mathcal{A}$'s capability of correctly guessing $\mathsf{gl}(x,r)$ into a way of guessing the i^{th} bit of x with probability that is non-negligibly greater than $1/2$. This suffices because given such a non-negligible success in guessing x_i, it is possible to boost this success further to $1-1/2n$, which suffices.

The proof of this case (where $\mathcal{A}$ guesses $\mathsf{gl}(x,r)$ with probability $1/2+\varepsilon(n)$) is significantly more involved than in the case of Proposition 6.13 (where $\mathcal{A}$ guesses $\mathsf{gl}(x,r)$ with probability $3/4+\varepsilon(n)$). The reason for this is that if $\mathcal{A}$ guesses $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x,r\oplus e^i)$ correctly with probability only $1/2+\varepsilon(n)$ each, then the probability that at least one is incorrect can only be upper-bound by $1/2-\varepsilon(n)+1/2-\varepsilon(n)=1-2\varepsilon(n)$. Thus, there is no guarantee that the majority of the guesses made by $\mathcal{A}'$ will be correct (when using the procedure from above).[3] We therefore must somehow compute $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x,r\oplus e^i)$

[3]We stress again that the events of successfully guessing $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x,r\oplus e^i)$ are not independent. Furthermore, we don't know that the minority guess will be correct; rather, we know nothing at all.

without invoking $\mathcal{A}$ twice. The way we do this is to invoke $\mathcal{A}$ on $\mathsf{gl}(x, r \oplus e^i)$ and "guess" the value $\mathsf{gl}(x, r)$ ourselves. This guess is generated in a special way so that the probability of the guess being correct (for all i) is reasonable good. (Of course, a naive way of guessing would be correct with only negligible probability, because we need to guess $\mathsf{gl}(x, r)$ for a polynomial number of r's.) The strategy for generating the guesses is via *pairwise independent* sampling. As we have already seen, Chebyshev's inequality can be applied to this case in order to bound the deviation from the expected value.

Continuing with this discussion, we show how the pairwise independent r's are generated. In order to generate m different r's (where m will be polynomial in n – see below), we select $\lceil l = \log(m+1) \rceil$ independent uniformly distributed strings in $\{0,1\}^n$; denote them by $s^1, \ldots, s^l$. Then, for every possible non-empty subset $I \subseteq \{1, \ldots, l\}$, we define $r^I = \oplus_{i \in I} s^i$. Notice that there are $2^l - 1$ non-empty subsets, and therefore we have defined $2^{\log(m+1)} - 1 = m$ different strings. Furthermore, each string is uniformly distributed when considered in isolation. We now claim that all of the strings r^I are pairwise independent. In order to see this, notice that for every two subsets $I \neq J$, there exists an index j such that $j \notin I \cap J$. Without loss of generality, assume that $j \in J$. Then, given r^I, it is clear that r^J is uniformly distributed because it contains a uniformly distributed string s^j that does not appear in r^I. Likewise, r^I is uniformly distributed given r^J because s^j "hides" r^I. (A formal proof of pairwise independence is straightforward and is omitted.) We now have the following two important observations:

1. Given the correct values $\mathsf{gl}(x, s_1), \ldots, \mathsf{gl}(x, s_l)$ and any non-empty subset $I \subseteq \{1, \ldots, l\}$, it is possible to correctly compute $\mathsf{gl}(x, r^I)$. This is due to the fact that by the definition of the predicate, $\mathsf{gl}(x, r^I) = \mathsf{gl}(x, \oplus_{i \in I} s^i) = \oplus_{i \in I} \mathsf{gl}(x, s^i)$.

2. The values $\mathsf{gl}(x, s_1), \ldots, \mathsf{gl}(x, s_l)$ can be correctly guessed with probability $1/2^l$ (this is the case because there are only l bits and so only $2^l = m + 1$ possibilities – one of which is correct). Note that since m is polynomial in n, it follows that 2^l is polynomial in n. Thus, the values $\mathsf{gl}(x, s_1), \ldots, \mathsf{gl}(x, s_l)$ can be correctly guessed with non-negligible probability.

Combining the above, we obtain the surprising property that this procedure yields a way of obtaining $m = \mathsf{poly}(n)$ pairwise independent strings $r \in \{0,1\}^n$ along with their corresponding *correct* $\mathsf{gl}(x, r)$ values, with non-negligible probability. It follows that these r and $\mathsf{gl}(x, r)$ values can then be used to compute x_i in the same way as in the proof of Proposition 6.13. Details follow.

The inversion algorithm $\mathcal{A}'$. We now provide a full description of the algorithm $\mathcal{A}'$ that receives an input y and uses algorithm $\mathcal{A}$ in order to find $f^{-1}(y)$. Upon input y, $\mathcal{A}'$ sets $n = |y|$ and $l = \lceil \log(2n/\varepsilon(n)^2 + 1) \rceil$, and proceeds as follows:

1. Uniformly choose $s^1, \ldots, s^l \leftarrow \{0,1\}^n$ and $\sigma^1 \ldots, \sigma^l \leftarrow \{0,1\}$ (σ^i is a guess for $\mathsf{gl}(x, s^i)$).
2. For every non-empty subset $I \subseteq \{1, \ldots, l\}$, define $r^I = \oplus_{i \in I}\ s^i$ and compute $\tau^I = \oplus_{i \in I}\ \sigma^i$ (τ^I is a guess for $\mathsf{gl}(x, r^I)$). We remark that as long as the σ^i values are all correct, so too are the τ^I values.
3. For every $i \in \{1, \ldots, n\}$, obtain a guess for x_i as follows:
   (a) For every non-empty subset $I \subseteq \{1, \ldots, l\}$, set $v_i^I = \tau^I \oplus \mathcal{A}(y, r^I \oplus e^i)$.
   (b) Guess $x_i = \mathsf{majority}_I\{v_i^I\}$ (i.e., take the bit that appeared a majority of the times in the previous step).
4. 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 $x \in f^{-1}(y)$. Before proceeding with the formal analysis, we provide an intuitive explanation. First, consider the case that the τ^I's are all correct (recall that this occurs with non-negligible probability). In such a case, we have that $v_i^I = x_i$ with probability at least $1/2 + \varepsilon(n)/2$ (this is due to the fact that $\mathcal{A}$ is invoked only once in computing v_i^I; the τ^I factor is already assumed to be correct). It therefore follows that a majority of the v_i^I values will equal the real value of x_i. Our analysis will rely on Chebyshev's inequality for the case of pairwise independent variables, because we need to compute the probability that the majority equals the correct x_i, where this majority is due to all the pairwise independent r^I's. We now present the formal proof.

CLAIM 6.18 *Assume that for every I, $\tau^I = \mathsf{gl}(x, r^I)$. Then, for every $x \in S_n$ and every $1 \leq i \leq n$, the probability that the majority of the v_i^I values equal x_i is at least $1 - 1/2n$. That is,*

$$\Pr\left[\left|\left\{I : \mathsf{gl}(x, r^I) \oplus \mathcal{A}(f(x), r^I \oplus e^i) = x_i\right\}\right| > \frac{1}{2} \cdot \left(2^l - 1\right)\right] > 1 - \frac{1}{2n}$$

PROOF For every I, define a 0-1 random variable X^I such that $X^I = 1$ if and only if $\mathcal{A}(y, r^I \oplus e^i) = \mathsf{gl}(x, r^I \oplus e^i)$. Notice that if $X^I = 1$, then $\mathsf{gl}(x, r^I) \oplus \mathcal{A}(y, r^I \oplus e^i) = x_i$. Since each r^I and $r^I \oplus e^i$ are uniformly distributed in $\{0,1\}^n$ (when considered in isolation), we have that $\Pr[X^I = 1] = s(x)$, and so for $x \in S_n$, we have that $\Pr[X^I = 1] \geq 1/2 + \varepsilon(n)/2$ implying that $\mathsf{Exp}[X^I] \geq 1/2 + \varepsilon(n)/2$. Furthermore, we claim that all the X^I random variables are pairwise independent. This follows from the fact that the r^I values

are pairwise independent. (Notice that if r^I and r^J are truly independent, then clearly so are X^I and X^J. The same is true of pairwise independence.)

Let $m = 2^l - 1$ and let X be a random variable that is distributed the same as all of the X^I's. Then, using Chebyshev's inequality, we have:

$$\begin{aligned}
\Pr\left[\sum_I X^I \leq \frac{1}{2}\cdot m\right] &= \Pr\left[\frac{\sum_I mX^I}{m} - \frac{1}{2}\cdot m \leq 0\right] \\
&= \Pr\left[\frac{\sum_I mX^I}{m} - \frac{1}{2}\cdot m - \frac{\varepsilon(n)}{2}\cdot m \leq -\frac{\varepsilon(n)}{2}\cdot m\right] \\
&\leq \Pr\left[\left|\frac{\sum_I mX^I}{m} - \left(\frac{1}{2}+\frac{\varepsilon(n)}{2}\right)\cdot m\right| \geq m\cdot\frac{\varepsilon(n)}{2}\right] \\
&\leq \frac{\mathsf{Var}[mX]}{(m\cdot\varepsilon(n)/2)^2\cdot m} \\
&= \frac{m^2\mathsf{Var}[X]}{(\varepsilon(n)/2)^2\cdot m^3} \\
&= \frac{\mathsf{Var}[X]}{(\varepsilon(n)/2)^2\cdot m}
\end{aligned}$$

Since $m = 2^l - 1 = 2n/\varepsilon(n)^2$, it follows from the above that:

$$\begin{aligned}
\Pr\left[\sum_I X^I \leq \frac{1}{2}\cdot m\right] &= \frac{\mathsf{Var}[X]}{(\varepsilon(n)/2)^2\cdot 2n/\varepsilon(n)^2} \\
&= \frac{\mathsf{Var}[X]}{n/2} \\
&< \frac{1/4}{n/2} = \frac{1}{2n}
\end{aligned}$$

where $\mathsf{Var}[X] < 1/4$ because $\mathsf{Var}[X] = E[X^2] - E[X]^2 = E[X] - E[X]^2 = E[X](1-E[X]) = (1/2+\varepsilon(n)/2)(1/2-\varepsilon(n)/2) = 1/4 - \varepsilon(n)^2/4 < 1/4$. This completes the proof of the claim because $\sum_I X^I$ is exactly the number of correct v_i^I values. ■

By Claim 6.18, we have that *if* all of the τ^I values are correct, then each x_i computed by $\mathcal{A}'$ is correct with probability at least $1 - 1/2n$. By the union bound over the failure probability of $1/2n$ for each i, we have that if all the τ^I values are correct, then the entire $x = x_1 \cdots x_n$ is correct with probability at least $1/2$. Notice now that the probability of the τ^I values being correct is independent of the analysis of Claim 6.18 and that this event happens with probability

$$\frac{1}{2^l} = \frac{1}{2n/\varepsilon(n)^2 + 1} > \frac{\varepsilon(n)^2}{4n}$$

Therefore, for $x \in S_n$, algorithm $\mathcal{A}'$ succeeds in inverting $y = f(x)$ with probability at least $\varepsilon(n)^2/8n$. Recalling that $|S_n| > \frac{\varepsilon(n)}{2} \cdot 2^n$, we have that $x \in S_n$ with probability $\varepsilon(n)/2$ and so the overall probability that $\mathcal{A}'$ succeeds in inverting $f(x)$ is greater than or equal to $\varepsilon(n)^3/16n$. Recalling that $\varepsilon(n) = 1/p(n)$ we have that for infinitely many n's, $\mathcal{A}'$ succeeds in inverting f with probability at least $p'(n) = 16n/p(n)^3$. Finally, noting that $\mathcal{A}'$ runs in polynomial-time, we obtain a contradiction to the one-wayness of f. ■

6.4 Constructions of Pseudorandom Generators

As we have seen in Section 3.3, pseudorandom generators are deterministic algorithms that receive a random input s of length n, and output a longer string of length $\ell(n)$ that looks random to any polynomial-time observer (or distinguisher). In this section we begin by showing how to construct pseudorandom generators that stretch the seed by one bit, under the assumption that one-way *permutations* exist. Then, we will show how to extend this to any polynomial expansion factor. Our presentation is based on one-way permutations. However, they can all be extended to hold for families of one-way permutations as well.

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 6.11). 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 6.19 *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 We have already seen the intuition and therefore begin directly with the proof. As with the theorems above, the proof is by reduction. That is, we show that if there exists a distinguisher D that can distinguish $G(s)$ from a truly random string, then we can use this distinguisher to construct

an adversary $\mathcal{A}$ that guesses $\mathsf{hc}(s)$ from $f(s)$ with probability that is non-negligibly greater than $1/2$.

Assume, by contradiction, that there exists a probabilistic polynomial-time distinguisher D and a non-negligible function $\varepsilon(n)$ such that

$$\left| \Pr_{s\in\{0,1\}^n}[D(f(s),\mathsf{hc}(s))=1] - \Pr_{r\in\{0,1\}^{n+1}}[D(r)=1] \right| \geq \varepsilon(n)$$

We call ε the "distinguishing gap" and say that D distinguishes $(f(s),\mathsf{hc}(s))$ from a random r with probability $\varepsilon(n)$.

As a first step to constructing an algorithm $\mathcal{A}$ to guess $\mathsf{hc}(s)$ from $f(s)$, we show that D can distinguish $(f(s),\mathsf{hc}(s))$ from $(f(s),\overline{\mathsf{hc}}(s))$ where $\overline{\mathsf{hc}}(s) = 1-\mathsf{hc}(s)$. In order to see this, first note that

$$\begin{aligned}&\Pr_{s\in\{0,1\}^n,\beta\in\{0,1\}}[D(f(s),\beta)=1]\\ &\quad = \frac{1}{2}\cdot\Pr[D(f(s),\mathsf{hc}(s))=1] + \frac{1}{2}\cdot\Pr[D(f(s),\overline{\mathsf{hc}}(s))=1]\end{aligned}$$

because with probability $1/2$ the random bit β equals $\mathsf{hc}(s)$, and with probability $1/2$ it equals $\overline{\mathsf{hc}}(s)$. Given this, we have:

$$\begin{aligned}&|\Pr[D(f(s),\mathsf{hc}(s))=1] - \Pr[D(f(s),\beta)=1]|\\ &\quad = |\Pr[D(f(s),\mathsf{hc}(s))=1] - \frac{1}{2}\cdot\Pr[D(f(s),\mathsf{hc}(s))=1]\\ &\qquad - \frac{1}{2}\cdot\Pr[D(f(s),\overline{\mathsf{hc}}(s))=1]|\\ &\quad = \frac{1}{2}|\Pr[D(f(s),\mathsf{hc}(s))=1] - \Pr[D(f(s),\overline{\mathsf{hc}}(s))=1]|\end{aligned}$$

where in all of the probabilities above $s\leftarrow\{0,1\}^n$ and $\beta\leftarrow\{0,1\}$ are chosen uniformly at random. By our contradicting assumption, and noting that $(f(s),\beta)$ is just a uniformly distributed string of length $n+1$, we have that

$$\begin{aligned}&\left|\Pr_{s\in\{0,1\}^n}[D(f(s),\mathsf{hc}(s))=1] - \Pr_{s\in\{0,1\}^n}[D(f(s),\overline{\mathsf{hc}}(s))=1]\right|\\ &\quad = 2\cdot\left|\Pr_{s\in\{0,1\}^n}[D(f(s),\mathsf{hc}(s))=1] - \Pr_{r\in\{0,1\}^{n+1}}[D(r)=1]\right|\\ &\quad \geq 2\varepsilon(n)\end{aligned}$$

Assume that $\Pr[D(f(s),\mathsf{hc}(s))=1] > \Pr[D(f(s),\overline{\mathsf{hc}}(s))=1]$; this is without loss of generality. We now use D to construct an algorithm $\mathcal{A}$ that is given $f(s)$ and guesses $\mathsf{hc}(s)$. Intuitively this is not difficult because D outputs 1 more often when it receives $(f(s),\mathsf{hc}(s))$ than when it receives $(f(s),\overline{\mathsf{hc}}(s))$. Upon input $y=f(s)$ for a random s, algorithm $\mathcal{A}$ works as follows:

1. Uniformly choose $\sigma\leftarrow\{0,1\}$

2. Invoke D upon (y, σ).

3. If D returns 1, then output σ. Otherwise, output $\overline{\sigma}$.

It remains to analyze the success probability of $\mathcal{A}$. As we have mentioned, $\mathcal{A}$ should succeed because D outputs 1 when $\sigma = \mathsf{hc}(s)$ with probability $2\varepsilon(n)$ more than it outputs 1 when $\sigma = \overline{\mathsf{hc}}(s)$. Formally,

$$\begin{aligned}
&\Pr[\mathcal{A}(f(s)) = \mathsf{hc}(s)] \\
&= \frac{1}{2}\Pr[\mathcal{A}(f(s)) = \mathsf{hc}(s) \mid \sigma = \mathsf{hc}(s)] + \frac{1}{2}\Pr[\mathcal{A}(f(s)) = \mathsf{hc}(s) \mid \sigma = \overline{\mathsf{hc}}(s)] \\
&= \frac{1}{2} \cdot \Pr[D(f(s), \mathsf{hc}(s)) = 1] + \frac{1}{2} \cdot \Pr[D(f(s), \overline{\mathsf{hc}}(s)) = 0]
\end{aligned}$$

where equality holds here because when D outputs 1, $\mathcal{A}$ outputs the value σ (and otherwise it outputs $\overline{\sigma}$). Thus, if $\sigma = \mathsf{hc}(s)$, we have that $\mathcal{A}$ invokes D on input $(f(s), \mathsf{hc}(s))$ and so $\mathcal{A}$ outputs $\mathsf{hc}(s)$ if and only if D outputs 1 upon input $(f(s), \mathsf{hc}(s))$. Likewise, if $\sigma \neq \mathsf{hc}(s)$, then $\mathcal{A}$ invokes D on input $(f(s), \overline{\mathsf{hc}}(s))$ and so $\mathcal{A}$ outputs $\mathsf{hc}(s)$ if and only if D outputs 0 upon input $(f(s), \overline{\mathsf{hc}}(s))$. Continuing the analysis, we have that

$$\begin{aligned}
&\Pr[\mathcal{A}(f(s)) = \mathsf{hc}(s)] \\
&\quad = \frac{1}{2} \cdot \Pr[D(f(s), \mathsf{hc}(s)) = 1] + \frac{1}{2}\left(1 - \Pr[D(f(s), \overline{\mathsf{hc}}(s)) = 1]\right) \\
&\quad = \frac{1}{2} + \frac{1}{2} \cdot \Pr[D(f(s), \mathsf{hc}(s)) = 1] - \frac{1}{2} \cdot \Pr[D(f(s), \overline{\mathsf{hc}}(s)) = 1] \\
&\quad = \frac{1}{2} + \frac{1}{2} \cdot \left(\Pr[D(f(s), \mathsf{hc}(s)) = 1] - \Pr[D(f(s), \overline{\mathsf{hc}}(s)) = 1]\right) \\
&\quad \geq \frac{1}{2} + \frac{1}{2} \cdot 2\varepsilon(n) = \frac{1}{2} + \varepsilon(n)
\end{aligned}$$

and so $\mathcal{A}$ guesses $\mathsf{hc}(s)$ with probability $1/2 + \varepsilon(n)$. Since $\varepsilon(n)$ is a non-negligible function, this contradicts the assumption that hc is a hard-core predicate of f. ■

6.4.2 Increasing the Expansion Factor

In this section, we show that the expansion factor of any pseudorandom generator can be increased by any polynomial amount. This means that the construction above (with expansion factor $\ell(n) = n + 1$) suffices for demonstrating the existence of pseudorandom generators with arbitrary polynomial expansion factor.

THEOREM 6.20 *If there exists a pseudorandom generator G_1 with expansion factor $\ell_1(n) = n + 1$, 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 G_1 is as follows. Given an initial seed s of length n, the generator G_1 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 G_1. The reason that these n bits can be used as a seed is because they are pseudorandom, and therefore essentially as good as a truly random seed. This procedure can be iteratively applied to output as many bits as desired; see Figure **??**.

We now formally describe the construction of G from G_1:

1. Let $s \in \{0,1\}^n$ be the seed, and denote $s_0 = s$.
2. For every $i = 1, \ldots, p(n)$, compute $(s_i, \sigma_i) = G_1(s_{i-1})$, where $\sigma_i \in \{0,1\}$ and $s_i \in \{0,1\}^n$.
3. Output $\sigma_1, \ldots, \sigma_{p(n)}$

We now proceed to prove that $G(s)$ is a pseudorandom string of length $p(n)$. We begin by proving this for the special and simple case of $p(n) = 2$. That is, the output of $G(s) = (\sigma_1, \sigma_2)$. Of course, this is not even a pseudorandom generator, because the output length is shorter than the input length. Nevertheless, it is helpful for understanding the basis of the proof.

A simplified case – $p(n) = 2$. Recall that the output (σ_1, σ_2) of $G(s)$ is derived in two stages: first by computing $(s_1, \sigma_1) = G_1(s)$ and then by computing $(s_2, \sigma_2) = G_1(s_1)$. Now, consider a "hybrid" experiment (the reason that we call this experiment "hybrid" will become apparent later) with an algorithm G' that receives $n+1$ random bits for input, denoted $\tilde{s} \in \{0,1\}^{n+1}$, and works as follows . Let $\tilde{s}^{[n]}$ be the first n bits of $\tilde{s}$ and let $\tilde{s}^{n+1}$ be the last bit of $\tilde{s}$. Then, G' works by computing $(s_2, \sigma_2) = G_1(\tilde{s}^{[n]})$ and setting $\sigma_1 = \tilde{s}^{n+1}$ (thus σ_1 is uniformly distributed). As with G, the algorithm G' outputs (σ_1, σ_2). First, notice that for every probabilistic polynomial-time distinguisher D there exists a negligible function negl such that

$$\left| \Pr_{s \in \{0,1\}^n}[D(G(s)) = 1] - \Pr_{\tilde{s} \in \{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1] \right| \leq \mathsf{negl}(n) \tag{6.3}$$

This must be the case because otherwise D can be used by an algorithm D' to distinguish $G_1(s)$ from random in the following way. Given a string w of length $n+1$, the algorithm D' can compute $(s_2, \sigma_2) = G_1(w^{[n]})$ and define $\sigma_1 = w^{n+1}$ (exactly like G'). Then, D' invokes D on (σ_1, σ_2) and outputs whatever D outputs. The important observations are as follows:

1. If $w = r$ is truly random, then the pair (σ_1, σ_2) prepared by D' is distributed identically to $G'(\tilde{s})$. Thus,

$$\Pr_{r \in \{0,1\}^{n+1}}[D'(r) = 1] = \Pr_{\tilde{s} \in \{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1]$$

2. If w is the output of $G_1(s)$ for a random $s \leftarrow \{0,1\}^n$, then the pair (σ_1, σ_2) prepared by D' is distributed identically to $G(s)$. In order to see this, note that in this case, σ_1 is the $n+1^{\text{th}}$ bit of $G_1(s)$ and σ_2 is the $n+1^{\text{th}}$ bit of $G_1(s_1)$, where $s_1 = G_1(s)^{[n]}$, exactly as in the construction of G. Therefore,

$$\Pr_{s\in\{0,1\}^n}[D'(G_1(s)) = 1] = \Pr_{s\in\{0,1\}^n}[D(G(s)) = 1]$$

Combining the above equations together, we have that

$$\left|\Pr_{s\in\{0,1\}^n}[D'(G_1(s)) = 1] - \Pr_{r\in\{0,1\}^{n+1}}[D'(r) = 1]\right| = \left|\Pr_{s\in\{0,1\}^n}[D(G(s)) = 1] - \Pr_{\tilde{s}\in\{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1]\right|$$

Now, if Equation (6.3) does not hold, then this implies that D' distinguishes $G_1(s)$ from random with non-negligible probability, in contradiction to the assumed pseudorandomness of G_1.

Next, we claim that for every probabilistic polynomial-time D there exists a negligible function negl such that

$$\left|\Pr_{\tilde{s}\in\{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1] - \Pr_{r\in\{0,1\}^2}[D(r) = 1]\right| \leq \mathsf{negl}(n) \tag{6.4}$$

This is proven in a very similar way as above. Specifically, if it does not hold, then D can be used by an algorithm D' to distinguish $G_1(s)$ from random in the following way. Given a string w of length $n+1$, the distinguisher D' sets σ_1 to be truly random and σ_2 to be the $n+1^{\text{th}}$ bit of w. As above, we have the following two observations:

1. If $w = r$ is truly random, then the pair (σ_1, σ_2) prepared by D' is truly random. Thus,

$$\Pr_{w\in\{0,1\}^{n+1}}[D'(w) = 1] = \Pr_{r\in\{0,1\}^2}[D(r) = 1]$$

2. If w is the output of $G_1(s)$ for a random $s \leftarrow \{0,1\}^n$, then the pair (σ_1, σ_2) prepared by D' is distributed identically to $G'(\tilde{s})$. This follows because G' sets σ_2 to be the $n+1^{\text{th}}$ bit of the output of $G_1(\tilde{s})$ and σ_1 to be truly random. Therefore,

$$\Pr_{s\in\{0,1\}^n}[D'(G_1(s)) = 1] = \Pr_{\tilde{s}\in\{0,1\}^n}[D(G'(\tilde{s})) = 1]$$

Combining the above equations together, we have that

$$\left|\Pr_{s\in\{0,1\}^n}[D'(G_1(s)) = 1] - \Pr_{w\in\{0,1\}^{n+1}}[D'(w) = 1]\right| = \left|\Pr_{\tilde{s}\in\{0,1\}^{n+1}}[D(G'(\tilde{s})) = 1] - \Pr_{r\in\{0,1\}^2}[D(r) = 1]\right|$$

Now, if Equation (6.4) does not hold, then this implies that D' distinguishes $G_1(s)$ from random with non-negligible probability, in contradiction to the assumed pseudorandomness of G_1.

Finally, combining Equations (6.3) and (6.4), we have that for every probabilistic polynomial-time distinguisher there exists a negligible function negl such that

$$\left| \Pr_{s\in\{0,1\}^n}[D(G(s))=1] - \Pr_{r\in\{0,1\}^2}[D(r)=1] \right| \leq \mathsf{negl}(n)$$

and so we have proven that G is pseudorandom (however, as we have mentioned, it is not a generator because it only outputs 2 bits).

*** The full proof.** The full proof of the theorem works in a similar way as above, except that we need to consider an arbitrary polynomial number of invocations of G_1. This proof uses a very important proof technique, called a *hybrid argument*, that is common in proofs in cryptography. As usual, the proof is by reduction and we show how a distinguisher for G with expansion factor $p(n)$ can be used to distinguish $G_1(s)$ from random. Assume by contradiction that there exists a probabilistic polynomial-time distinguisher D and a non-negligible function ε such that

$$\left| \Pr_{s\in\{0,1\}^n}[D(G(s))=1] - \Pr_{r\in\{0,1\}^{p(n)}}[D(r)=1] \right| = \varepsilon(n)$$

For every i, we define a *hybrid* random variable H_n^i as a string with a length i prefix that is truly random, and a length $p(n)-i$ suffix that is pseudorandom. Specifically, H_n^i is computed by first fixing its first i bits to be truly random. Then, a random $s \leftarrow \{0,1\}^n$ is chosen and $p(n)-i$ bits are obtained by iteratively applying $G_1(s)$ and taking the last bit each time (stated differently, the remaining $p(n)-i$ bits are taken as $G(s)$ with expansion factor $\ell(n) = p(n)-i$). Clearly, this random variable is a "hybrid" between a random string and a pseudorandom one.

Notice that $H_n^0 = G(s)$, because all of the bits are taken from $G(s)$ with expansion factor $p(n)$, and that $H_n^{p(n)}$ is a uniformly distributed string of length $p(n)$. 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). However, the difference between one hybrid and its neighbor is a single application of the pseudorandom generator. Thus, as we will show, the distinguisher D for G can be transformed into a distinguisher D' for G_1.

We now describe the probabilistic polynomial-time distinguisher D' that uses D and distinguishes between $G_1(s)$ and r, where $s \leftarrow \{0,1\}^n$ and $r \leftarrow \{0,1\}^{n+1}$. Upon input $w \in \{0,1\}^{n+1}$, distinguisher D' chooses a random $i \leftarrow \{0,\ldots,p(n)-1\}$ and constructs a string $\tilde{s}$ as follows. It first chooses i random bits and sets them to be the first i bits of $\tilde{s}$. Then, it takes the

$n + 1^{\text{th}}$ bit of w and sets it to be the $i + 1^{\text{th}}$ bit of $\tilde{s}$. Finally, it computes $G(w)$ with expansion factor $\ell(n) = p(n) - i - 1$ and writes the result as the remaining bits of $\tilde{s}$. Finally, D' invokes D on $\tilde{s}$ and outputs whatever D does. The important observations here are as follows:

1. If $w = G_1(s)$ for some random $s \leftarrow \{0,1\}^n$, then $\tilde{s}$ is distributed exactly like H_n^i (because the first i bits are random and the rest are pseudorandom). Furthermore, each i is chosen with probability exactly $1/p(n)$. Thus,

$$\Pr_{s \in \{0,1\}^n}[D'(G_1(s)) = 1] = \frac{1}{p(n)} \cdot \sum_{i=0}^{p(n)-1} \Pr[D(H_n^i) = 1]$$

2. If $w = r$ is a truly random string ($r \leftarrow \{0,1\}^{n+1}$), then $\tilde{s}$ is distributed exactly like H_n^{i+1} (because the first $i+1$ bits are random and the rest are pseudorandom). Thus,

$$\Pr_{r \in \{0,1\}^{n+1}}[D'(r) = 1] = \frac{1}{p(n)} \cdot \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1].$$

Combining this together, and using the fact that

$$\left| \sum_{i=0}^{p(n)-1} \Pr[D(H_n^i) = 1] - \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1] \right| = \left| \Pr[D(G(H_n^0)) = 1] - \Pr[D(H_n^{p(n)}) = 1] \right|$$

(since it is a telescopic sum), we have that:

$$\begin{aligned}
&\left| \Pr_{s \in \{0,1\}^n}[D'(G_1(s)) = 1] - \Pr_{r \in \{0,1\}^{n+1}}[D'(r) = 1] \right| \\
&\quad = \frac{1}{p(n)} \cdot \left| \sum_{i=0}^{p(n)-1} \Pr[D(H_n^i) = 1] - \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1] \right| \\
&\quad = \frac{1}{p(n)} \cdot \left| \Pr[D(H_n^0) = 1] - \Pr[D(H_n^{p(n)}) = 1] \right| \\
&\quad = \frac{1}{p(n)} \cdot \left| \Pr_{s \in \{0,1\}^n}[D(G(s)) = 1] - \Pr_{r \in \{0,1\}^{n+1}}[D(r) = 1] \right| \\
&\quad = \frac{\varepsilon(n)}{p(n)}
\end{aligned}$$

which is non-negligible, in contradiction to the pseudorandomness of G_1. ■

The hybrid technique. The hybrid proof technique is used in many proofs of cryptographic constructions and is considered a basic technique. Note that there are three conditions for using it. First, the extreme hybrids are the same as the original distributions (in this case, $H_n^0 = G(s)$ and $H_n^{p(n)} = r \in \{0,1\}^{p(n)}$). Second, the capability of distinguishing neighboring hybrids can be translated into the capability of distinguishing the underlying primitive (in this case, $G_1(s)$ from $r \in \{0,1\}^{n+1}$). Finally, the number of hybrids is polynomial (and so the degradation of distinguishing success is only polynomial). We stress that although it may seem strange that we invoke the distinguisher D on an input that it does not expect, D is just an algorithm and so its behavior is well-defined (irrespective of what it "means" to do).

An efficiency improvement. It is possible to modify the construction of the generator G given in the proof of Theorem 6.20 so that G outputs $s_{p(n)}$ as well as $\sigma_1, \ldots, \sigma_{p(n)}$. The proof of this modification can be obtained from the proof of Theorem 6.20 and we leave it as an exercise.

An explicit generator with arbitrary expansion factor. By combining the construction of Theorem 6.19 (that states that $G(s) = (f(s), \mathsf{hc}(s))$ is a pseudorandom generator) together with the proof of Theorem 6.20 (actually, with the efficiency improvement described above), we obtain that for every polynomial p,

$$G_\ell(s) = \left(f^{p(n)}(s), \mathsf{hc}(s), \mathsf{hc}(f(s)), \ldots, \mathsf{hc}(f^{p(n)-n}(s))\right)$$

is a pseudorandom generator with expansion factor $p(n)$, assuming that f is a one-way permutation and hc is a hard-core predicate of f.

Modern (heuristic) stream cipher design. Many modern stream ciphers (i.e., pseudorandom generators) work by maintaining a large *pseudorandom* internal state. In each iteration of the generator, some pseudorandom bits are output and the internal state is updated. It is interesting to note that the construction described in Theorem 6.20 (and in Figure **??**) works in this exact way. This may be seen as evidence that this is a good heuristic, since in some circumstances it can be used to achieve a provable-secure construction.

6.5 Constructions of 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).

Before presenting the full construction, we motivate it through 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, $G_0(s)$ is the first half of the output and $G_1(s)$ is the second half of the output. We now use G to construct a keyed function (with a key of length n bits) that takes a *single input bit* and outputs strings of length n bits. For a randomly-chosen key k, we define:

$$F_k(0) = G_0(k) \qquad F_k(1) = G_1(k)$$

We claim now that this function is pseudorandom.[4] This follows immediately from the fact that G is a pseudorandom generator and so no polynomial-time algorithm can distinguish $G(k) = (G_0(k), G_1(k))$ from a truly random string. Now, a truly random function f of this type has two random strings, one defining $f(0)$ and the other defining $f(1)$. Thus, these two functions cannot be distinguished by any polynomial-time algorithm.

We take this construction a step further and define a pseudorandom function that has a *two-bit input* and an n-bit output. Namely, for a random k, define:

$$F_k(00) = G_0(G_0(k)) \quad F_k(10) = G_1(G_0(k))$$
$$F_k(01) = G_0(G_1(k)) \quad F_k(11) = G_1(G_1(k))$$

We 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 all pseudorandom, *even when viewed all together*. (As above, this suffices to prove that the function F_k is pseudorandom.) In order to see that all four strings are indeed pseudorandom, consider the following *hybrid distribution*:

$$G_0(k_0),\ G_0(k_1),\ G_1(k_0),\ G_1(k_1)$$

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.

[4] Note that our definition of pseudorandom function (Definition 3.24) restricts the function to domain and range of $\{0,1\}^n$. Nevertheless, the definition can be extended to domain $\{0,1\}$ and range $\{0,1\}^n$ in a straightforward way.

As we have discussed, long pseudorandom strings yield pseudorandom functions. This is due to the fact that a truly random function is just a very long random string, where a different part of the string is allocated as output for every possible input. (In order to see this, one can view a random function as just a large table of random values.) The fundamental leap between pseudorandom generators and pseudorandom functions is due to the fact that when the function is defined over inputs of length n, the string defining the function is of length $n2^n$, which is exponential and so cannot be computed.[5] Despite this fact, the above ideas can be used in order to obtain a pseudorandom function. Specifically, the full construction below works in the same way as above, except that the pseudorandom generator is applied n times, once for each input bit.

CONSTRUCTION 6.21 Pseudorandom function.

Let G be a deterministic function that maps inputs of length n into outputs of length $2n$ (G will be instantiated as a pseudorandom generator with $\ell(n) = 2n$). Denote by $G_0(k)$ the first n bits of G's output, and by $G_1(k)$ the second n bits of G's output. For every $k \in \{0,1\}^n$, define the function $F_k : \{0,1\}^n \to \{0,1\}^n$ as:

$$F_k(x_1 x_2 \cdots x_n) = G_{x_n}(\cdots(G_{x_2}(G_{x_1}(k)))\cdots)$$

The function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ is obtained by defining $F(k,x) = F_k(x)$ for every k and x of the same length (and setting the output to be $\perp$ if k and x are not of the same length).

This construction can be viewed as a full binary tree of depth n, defined as follows. The value at the root equals the key k. Then, for any node of value k', the left son of k' has value $G_0(k')$ and the right son of k' has value $G_1(k')$. The function on an input value $x = x_1 \cdots x_n$ 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 inputs to the function are also of length n *exactly,* and thus only values in the leaves are output (if the values at internal nodes could also be output, the result would not be a pseudorandom function). Note also that the size of the tree is exponential in n; in particular, there are 2^n leaves. Nevertheless, we never need to construct and hold the tree explicitly. Rather, the values on the path (and so the value of the appropriate leaf) can be efficiently obtained

[5] Indeed, a pseudorandom function from two input bits to n output bits could have been constructed by using a pseudorandom generator with $\ell(n) = 4n$, and allocating a different n-bit portion to each of the 4 inputs. However, such a strategy would *not* generalize to pseudorandom functions with domain $\{0,1\}^n$.

given the key k, and so can be computed dynamically. See Figure **??** for a graphical presentation of the construction.

We now claim that when G is a pseudorandom generator, the above construction constitutes a pseudorandom function.

THEOREM 6.22 *If the function G is a pseudorandom generator with expansion factor $\ell(n) = 2n$, then Construction 6.21 is an efficiently computable pseudorandom function.*

PROOF (Sketch) The intuition behind the proof of this theorem follows from the motivating examples given above. Namely, the hybrid distribution provided for the case of a 2-bit input can be extended to inputs of length n. This extension works by just continuing to build the binary tree of Figure **??** (note that the case of 1 input bit gives the root and the next level of the tree and the case of 2 input bits extends this to an additional level of the tree). The reason why it is possible to continue extending the construction is that if the intermediate result is pseudorandom, then it can be replaced by truly random strings, that can then be used as seeds to the pseudorandom generator once again.

The actual proof of the theorem works by a hybrid argument (see the proof of Theorem 6.20), and we only sketch it here. We define a hybrid random variable H_n^i to be a full binary tree of depth n where the nodes of levels 0 to i are labelled with independent truly random values, and the nodes of levels $i+1$ to n are constructed as in Construction 6.21 (given the labels of level i). We note that in H_n^i, the labels in nodes 0 to $i-1$ are actually irrelevant. The function associated with this tree is obtained as in Construction 6.21 by outputting the appropriate values in the leaves.

Notice that H_n^n is a truly random function, because all of the leaves are given truly random and independent values. On the other hand, H_n^0 is exactly Construction 6.21 (because only the key is random and everything else is pseudorandom, as in the construction). Using a standard hybrid argument as made in the proof of Theorem 6.20, we obtain that if a polynomial-time distinguisher D can distinguish Construction 6.21 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 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 one more time on the way from the root to the leaves of the tree. The actual proof is more tricky 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 6.21 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 (filling all values to that point with arbitrary values – they are of no consequence). Then, D' takes one of its input samples (of length $2n$) and labels the left son of the reached node with the first half of the sample and the right son with the second half of the sample. D' then continues to compute the output as in Construction 6.21. 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' works by filling 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. (Notice that the seeds to these pseudorandom values are not known to D' but this makes no difference to the result.)

By carrying out a similar hybrid analysis as in the proof of Theorem 6.20, we obtain that if D distinguishes Construction 6.21 from a truly random function with non-negligible 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 $\varepsilon(n)/n$. Since $\varepsilon(n)$ is non-negligible, this contradicts the assumption that G is a pseudorandom generator. ■

6.6 Constructions of Pseudorandom Permutations

In this section, we show how pseudorandom permutations can be constructed from pseudorandom functions. Recall that a pseudorandom permutation is a pseudorandom function F with the property that for every $k \in \{0,1\}^n$, the function $F_k(\cdot)$ is a bijection over $\{0,1\}^n$. In addition, the

security requirement is that it is hard to distinguish such a function from a truly random one, even when given oracle access to both F_k and F_k^{-1} (i.e., the inversion function). See Section 3.6.3 for more details. In contrast to the previous sections in this chapter, we will *not* prove the security of this construction (nor even provide a proof sketch) and refer the interested reader to more advanced texts.

Feistel structures revisited. Recall that a Feistel structure is a way of constructing an invertible function from non-invertible operations (see Section 5.2). In some sense, this is exactly what we wish to do here. Namely, given a pseudorandom function, we wish to construct another pseudorandom function that is a bijection (and thus is invertible). In another interesting connection between theoretical and heuristic constructions, it turns out that *four rounds* of a Feistel structure with the F-functions taken as pseudorandom functions, constitutes a pseudorandom permutation. That is, we have the following construction:

CONSTRUCTION 6.23 Pseudorandom permutation.

Let F be a pseudorandom function so that for every n and every $k \in \{0,1\}^n$, the function $F_k(\cdot)$ maps inputs of length n into outputs of length n. Define the function F' that maps inputs of length $2n$ into outputs of length $2n$ as follows:

- **Inputs:** a key $k \in \{0,1\}^{4n}$ of length $4n$ and an input $x \in \{0,1\}^{2n}$; denote $x = (x_1, x_2)$ where $|x_1| = |x_2| = n$
- **Computation:**
  1. Denote $k = (k_1, k_2, k_3, k_4)$ where each k_i is of length n
  2. Compute $\alpha_1 = x_2$ and $\alpha_2 = x_1 \oplus F_{k_1}(x_2)$
  3. Compute $\beta_1 = \alpha_2$ and $\beta_2 = \alpha_1 \oplus F_{k_2}(\alpha_2)$
  4. Compute $\gamma_1 = \beta_2$ and $\gamma_2 = \beta_1 \oplus F_{k_3}(\beta_2)$
  5. Compute $y_1 = \gamma_2$ and $y_2 = \gamma_1 \oplus F_{k_4}(\gamma_2)$
  6. Output (y_1, y_2)

We remark that Construction 6.23 deviates slightly from Definition 3.28 because the input and key lengths are different. Nevertheless, Definition 3.28 can easily be modified to allow this. Notice that in each round of the Feistel structure, a different key k_i is used. Furthermore, the construction is "legal" because F_{k_i} with a key of length n is applied to an input that is also of length n. See Figure **??** for a graphical presentation of the construction.

We have the following theorem:

THEOREM 6.24 *If the function F is pseudorandom and maps n-bit strings to n-bit strings, then Construction 6.23 is an efficiently computable and efficiently invertible pseudorandom function that maps $2n$-bit strings to $2n$-bit strings (and uses a key of length $4n$).*

As we have mentioned, we will not prove the theorem. However, we do remark that it is easy to prove that the resulting function is indeed a bijection and that it is efficiently invertible. We leave this for an exercise. We also remark that if only *three rounds* of a Feistel is used, then the result is a weaker type of pseudorandom permutation where the distinguisher is only given access to the function oracle (and not the inversion oracle).

6.7 Private-Key Cryptography – Necessary and Sufficient Assumptions

Summing up what we have seen so far in this chapter, we have the following:

1. For every one-way function f, there exists a hard-core predicate for the function $g(x,r) = (f(x),r)$.
2. If there exist one-way permutations, then there exist pseudorandom generators (the proof of this fact uses the hard-core predicate that is guaranteed to exist).
3. If there exist pseudorandom generators, then there exist pseudorandom functions.
4. If there exist pseudorandom functions, then there exist pseudorandom permutations.

Thus, pseudorandom generators and functions (that can be used for encryption) can be achieved assuming the existence of one-way permutations. In actuality, it is possible to construct pseudorandom generators from *any* one-way function (we did not present this construction as it is highly complex). Thus, we have the following fundamental theorem:

THEOREM 6.25 *If there exist one-way functions, then there exist pseudorandom generators, pseudorandom functions and pseudorandom permutations.*

Recall now that all of the private-key cryptographic tools that we have studied so far can be constructed from pseudorandom generators and functions. This includes basic encryption that is secure for eavesdropping adversaries,

more advanced methods that are CPA and CCA-secure, and message authentication codes. Noting that CCA-security implies both CPA and eavesdropping security, we have the following theorem:

THEOREM 6.26 *If there exist one-way functions, then there exist encryption schemes that have indistinguishable encryptions under chosen-ciphertext attacks, and there exist message authentication codes that are existentially unforgeable under chosen message attacks.*

Stated informally, *one-way functions are a sufficient assumption for private-key cryptography*. Given this fact, a question of great importance is whether or not one-way functions are also a *necessary* (or minimal) assumptions. We now show that this is indeed the case.

Pseudorandom entities imply one-way functions. We begin by showing that the existence of pseudorandom generators and functions imply the existence of one-way functions. In fact, since it is easy to show that pseudorandom functions imply pseudorandom generators (just define $G(s) = (F_s(0), F_s(1))$), it follows that it suffices to prove the following:

PROPOSITION 6.27 *If there exist pseudorandom generators, then there exist one-way functions.*

PROOF We prove this proposition by showing how to construct a one-way function f from a pseudorandom generator G. The construction is simple and is defined by $f(x) = G(x)$. In order to show that f is indeed one-way, we need to show that it can be efficiently computed but that it is hard to invert. Efficient computability is straightforward (by the fact that G can be computed in polynomial-time). Regarding inversion, we will show that the ability to invert f can be translated into the ability to distinguish G from random. Intuitively, this holds because the ability to invert f equals the ability to find the seed used by the generator. (Recall that a one-way function is invoked on a uniformly distributed string, just like a pseudorandom generator.)

Formally, let G be a pseudorandom generator with expansion factor $\ell(n) = 2n$ (recall that the expansion factor can be increased, so assuming $\ell(n) = 2n$ is without loss of generality). Assume by contradiction that $f(x) = G(x)$ is not a one-way function. This implies that there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a non-negligible function ε such that

$$\Pr\left[\mathcal{A}(f(x)) \in f^{-1}(f(x))\right] = \Pr\left[\mathcal{A}(G(x)) \in G^{-1}(G(x))\right] \geq \varepsilon(n)$$

where $x \leftarrow \{0,1\}^n$ is uniformly chosen. We now use $\mathcal{A}$ to construct a distinguisher D for G. The distinguisher D receives a string $w \in \{0,1\}^{2n}$ (that is either pseudorandom or truly random) and runs $\mathcal{A}$ on w. Then, D outputs 1

if and only if $\mathcal{A}$ outputs a value x such that $G(x) = w$. We claim that D is a good distinguisher for G. We first show that

$$\Pr_{r\in\{0,1\}^{2n}}[D(r) = 1] \leq \frac{1}{2^n}$$

This holds because there are at most 2^n values w in the range of G; namely the values $\{G(s)\}_{s\in\{0,1\}^n}$. However, there are 2^{2n} different values r in $\{0,1\}^{2n}$. It therefore follows that a uniformly distributed string of length $2n$ is in the range of G with probability at most $2^n/2^{2n} = 2^{-n}$. Of course, if w is not even in the range of G, then $\mathcal{A}$ cannot output x such that $G(x) = w$ and so D definitely outputs 0. Next, we claim that

$$\Pr_{s\in\{0,1\}^n}[D(G(s)) = 1] \geq \varepsilon(n)$$

This follows immediately from the way we constructed D and from the assumed success of $\mathcal{A}$ in inverting the function $f(x) = G(x)$. We therefore have that

$$|\Pr_{r\in\{0,1\}^{2n}}[D(r) = 1] - \Pr_{s\in\{0,1\}^n}[D(G(s)) = 1]| \geq \varepsilon(n) - \frac{1}{2^n}$$

Since ε is non-negligible, the function $\varepsilon(n)-2^{-n}$ is also non-negligible. We conclude that D distinguishes the output of G from random with non-negligible probability, in contradiction to the assumed pseudorandomness of G. Thus, we conclude that f *is* a one-way function and so the existence of pseudorandom generators implies the existence of one-way functions. ■

Private-key encryption schemes imply one-way functions. Proposition 6.27 tells us that if we want to build pseudorandom generators or functions, then we need to assume that one-way functions exist. It is important to note that this does *not* imply that one-way functions are needed for constructing secure private-key encryption schemes (it may be possible to construct secure encryption schemes in a completely different way). Furthermore, it *is* possible to construct perfectly secret encryption schemes (see Chapter 2), as long as the plaintext is shorter than the key. Thus, the proof that secure private-key encryption implies one-way function is somewhat more subtle. Since security in the presence of eavesdropping adversaries is the weakest of the definitions we considered, we prove the proposition based on such schemes.

PROPOSITION 6.28 *If there exist private-key encryption schemes that have indistinguishable encryptions in the presence of eavesdropping adversaries (as in Definition 3.9), then there exist one-way functions.*

PROOF We remark that since Definition 3.9 is considered, security must hold for messages of any length output by the adversary. In particular, security must hold for messages that are longer than the key, and so perfectly

secret encryption schemes do not meet the requirements. This is important because perfectly-secret encryption (with the key at least as long as the plaintext) can be constructed without any assumptions, and in particular without assuming the existence of one-way functions. The idea behind the proof here is similar to that of Proposition 6.27. Specifically, we define a one-way function for which successful inversion implies breaking the encryption scheme (specifically, finding the secret key). Then, we show how any adversary that succeeds in inverting the one-way function can be used to distinguish encryptions (in contradiction to the assumed security of the encryption scheme). In this proof we assume familiarity with Definition 3.9 and $\mathsf{PrivK}^{\mathsf{eav}}$ (see Section 3.2.1).

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 Enc uses at most ℓ bits of randomness in order to encrypt a plaintext of length ℓ. (The proof can be modified to hold for any polynomial number of random bits. We assume this for simplicity.) Denote an encryption of a message m with key k and random coins r by $\mathsf{Enc}_k(m;r)$. Now, define a function $f(k,m,r) = (\mathsf{Enc}_k(m;r), m)$ where m is twice the length of k (i.e., $|m| = 2|k|$). Thus, if the input length is n, we have that k is of length $n/5$ and m are r are both of length $2n/5$. 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. Assume by contradiction that there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a non-negligible function ε such that

$$\Pr[\mathcal{A}(f(x)) \in f^{-1}(f(x))] \geq \varepsilon(n)$$

where $x \leftarrow \{0,1\}^n$ is uniformly chosen. Recall, $x = (k,m,r)$ and $f(x) = f(k,m,r) = (\mathsf{Enc}_k(m;r), m)$. We use $\mathcal{A}$ to construct an adversary $\mathcal{A}'$ that succeeds with non-negligible advantage in $\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\frac{n}{5})$. (We note that the key length is $n/5$. Therefore, it will be convenient to refer to the security parameter in the encryption experiment as being of length $n/5$.) The adversary $\mathcal{A}'$ simply chooses two random strings $m_0, m_1 \leftarrow \{0,1\}^{2n/5}$ of length $2n/5$ and receives back $c = \mathsf{Enc}_k(m_b;r)$ for some $b \in \{0,1\}$ and $r \in \{0,1\}^{2n/5}$. Then, $\mathcal{A}'$ invokes $\mathcal{A}$ on (c, m_0). If $\mathcal{A}$ outputs a tuple (k', m_0, r') such that $c = \mathsf{Enc}_{k'}(m_0;r')$, then $\mathcal{A}'$ outputs $b' = 0$. Otherwise, $\mathcal{A}$ outputs a uniformly chosen $b' \leftarrow \{0,1\}$. We now show that $\mathcal{A}'$ guesses the correct value $b' = b$ with probability that is non-negligibly greater than $1/2$.

We first claim that if $c = \mathsf{Enc}_k(m_0;r)$ for some r, then $\mathcal{A}$ outputs (k', m_0, r') such that $c = \mathsf{Enc}_{k'}(m_0;r')$ with probability at least $\varepsilon(n)$. This follows immediately from the assumption regarding $\mathcal{A}$'s capability of inverting f. Furthermore, in such a case, $\mathcal{A}'$ always succeeds in $\mathsf{PrivK}^{\mathsf{eav}}$. This is due to the fact that $\mathcal{A}'$ outputs 0, and we are considering the case that $c = \mathsf{Enc}_k(m_0)$ and so $b = 0$. We note that k' may not be the "correct key" (i.e., the key k chosen by the experiment may be such that $k \neq k'$, but nevertheless $\mathsf{Enc}_k(m_0;r) = c = \mathsf{Enc}_{k'}(m_0;r')$ for some r and r'). However, $\mathcal{A}'$ still outputs 0 and $b = 0$; thus $\mathcal{A}'$ succeeds. Denoting by $\mathsf{invert}_{\mathcal{A}}$ the event that $\mathcal{A}$

outputs (k', m_0, r') where $c = \mathsf{Enc}_{k'}(m_0; r')$, we have that:

$$\Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 0\right] \geq \varepsilon(n)$$

and

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid \mathsf{invert}_{\mathcal{A}} \ \wedge \ b = 0\right] = 1$$

Notice also that if the event $\mathsf{invert}_{\mathcal{A}}$ does *not* occur, then $\mathcal{A}'$ succeeds in $\mathsf{PrivK}^{\mathsf{eav}}$ with probability exactly $1/2$ (because in this case $\mathcal{A}'$ just outputs a random bit). Thus,

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid \neg\mathsf{invert}_{\mathcal{A}} \ \wedge \ b = 0\right] = \frac{1}{2}$$

Combining the above, we have:

$$\begin{aligned}
&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid b = 0\right] \\
&= \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 0\right] \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid \mathsf{invert}_{\mathcal{A}} \ \wedge \ b = 0\right] \\
&\quad + \ \Pr\left[\neg\mathsf{invert}_{\mathcal{A}} \mid b = 0\right] \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid \neg\mathsf{invert}_{\mathcal{A}} \ \wedge \ b = 0\right] \\
&= \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 0\right] \cdot 1 + (1 - \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 0\right]) \cdot \frac{1}{2} \\
&= \frac{1}{2} \cdot \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 0\right] + \frac{1}{2} \\
&\geq \frac{1}{2} + \frac{\varepsilon(n)}{2}
\end{aligned}$$

We now proceed to analyze the case that $b = 1$. At first sight, it may seem that in this case, $\mathcal{A}'$ always succeeds in $\mathsf{PrivK}^{\mathsf{eav}}$ with probability $1/2$, because c is an encryption of m_1 and so cannot possibly be "inverted" by $\mathcal{A}$ to be an encryption of m_0. However, this is not true. For some $c = \mathsf{Enc}_k(m_1)$, there may exist a key k' such that $m_0 = \mathsf{Dec}_{k'}(c)$; indeed perfectly-secret encryption schemes always have this property. We will now show that this occurs (in this computational setting) with at most negligible probability. Fix k, m_1, r, and $c = \mathsf{Enc}_k(m_1; r)$. We upper-bound the probability that there exists a key k' such that $m_0 = \mathsf{Dec}_{k'}(c)$. Notice that k' is of length $n/5$ and m_0 is a uniformly distributed string of length $2n/5$. Furthermore, m_0 is chosen independently of m_1. Since m_0 is independent of m_1 and $c = \mathsf{Enc}_k(m_1)$, we can consider the probability that there exists a key k' such that $m_0 = \mathsf{Enc}_{k'}(c)$, where the probability is *over the random choice of* m_0. We have:

$$\begin{aligned}
\Pr_{m_0 \in \{0,1\}^{2n/5}}[\exists k' \ m_0 = \mathsf{Dec}_{k'}(c)] &\leq \sum_{k' \in \{0,1\}^{n/5}} \Pr_{m_0 \in \{0,1\}^{2n/5}}[m_0 = \mathsf{Dec}_{k'}(c)] \\
&= \sum_{k' \in \{0,1\}^{n/5}} \frac{1}{2^{2n/5}} \\
&= \frac{2^{n/5}}{2^{2n/5}} = \frac{1}{2^{n/5}}
\end{aligned}$$

where the first inequality is by the union bound, the second equality is due to the uniform choice of m_0 and the unique decryption requirement on the encryption scheme (i.e., for any given key k' there is only a single value $\mathsf{Dec}_{k'}(c)$), and the third equality is due to the number of different possible keys.[6] Now, if there does not exist such a key k', then clearly $\mathcal{A}$ cannot "invert" and output (k', m_0, r') such that $c = \mathsf{Enc}_{k'}(m_0; r')$. Thus,

$$\Pr\left[\neg\mathsf{invert}_{\mathcal{A}} \mid b = 1\right] \geq 1 - \frac{1}{2^{n/5}}$$

Furthermore, as in the case of $b = 0$, if the event $\mathsf{invert}_{\mathcal{A}}$ does *not* occur, then $\mathcal{A}'$ succeeds in $\mathsf{PrivK}^{\mathsf{eav}}$ with probability exactly $1/2$. Noting finally that if $\mathsf{invert}_{\mathcal{A}}$ *does* occur in this case then $\mathcal{A}'$ succeeds with probability 0 (because it outputs 0 when $b = 1$), we have:

$$\begin{aligned}
&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid b = 1\right] \\
&= \Pr\left[\mathsf{invert}_{\mathcal{A}} \mid b = 1\right] \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid \mathsf{invert}_{\mathcal{A}} \ \wedge \ b = 1\right] \\
&\quad + \ \Pr\left[\neg\mathsf{invert}_{\mathcal{A}} \mid b = 1\right] \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid \neg\mathsf{invert}_{\mathcal{A}} \ \wedge \ b = 1\right] \\
&= \Pr\left[\neg\mathsf{invert}_{\mathcal{A}} \mid b = 1\right] \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid \neg\mathsf{invert}_{\mathcal{A}} \ \wedge \ b = 1\right] \\
&\geq \left(1 - \frac{1}{2^{n/5}}\right) \cdot \frac{1}{2} \\
&= \frac{1}{2} - \frac{1}{2 \cdot 2^{n/5}}
\end{aligned}$$

Putting the above together (and noting that $b = 0$ and $b = 1$ with probability $1/2$), we have:

$$\begin{aligned}
&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1\right] \\
&= \frac{1}{2} \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid b = 0\right] + \frac{1}{2} \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(\tfrac{n}{5}) = 1 \mid b = 1\right] \\
&\geq \frac{1}{2}\left(\frac{1}{2} + \frac{\varepsilon(n)}{2}\right) + \frac{1}{2}\left(\frac{1}{2} - \frac{1}{2 \cdot 2^{n/5}}\right) \\
&= \frac{1}{2} + \frac{\varepsilon(n)}{4} - \frac{1}{4 \cdot 2^{n/5}}
\end{aligned}$$

Since ε is a non-negligible function (and $2^{-n/5}$ is negligible), it follows that $\mathcal{A}'$ succeeds with probability that is non-negligibly greater than $1/2$, in contradiction to the assumed security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ in the presence of eavesdropping adversaries. Thus, $f(k, m, r) = (\mathsf{Enc}_k(m; r), m)$ is a one-way function. ■

[6] This bound only holds when the plaintext can be longer than the key. Therefore the proof falls completely when one-time pad like schemes are considered.

Message authentication codes imply one-way functions. Having covered pseudorandom generators and functions and secure private-key encryption, it remains to show that message authentication codes also imply the existence of one-way functions. The proof of this fact is also somewhat subtle, because unconditionally secure message authentication codes do exist (again, with a limitation on the relation between the number of messages authenticated and the key length). We state this proposition without proof:

PROPOSITION 6.29 *If there exist message authentication codes that are existentially unforgeable under chosen message attacks, then there exist one-way functions.*

Conclusion and discussion. We conclude that the existence of one-way functions is both a necessary and sufficient assumption for achieving private-key cryptography. Thus the assumption regarding the existence of one-way functions is indeed *minimal* when it comes to private-key cryptography. We remark that this seems not to be the case for public-key encryption, that will be studied next. That is, although one-way functions is a necessary assumption also for public-key cryptography, it seems not to be a sufficient one. (We remark that in addition to 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 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 indeed exactly how we defined pseudorandom generators (and functions). 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 series of finite probability distributions (rather than being a single distribution). This formalism is a necessary by-product of the asymptotic approach because distinguishing two finite distributions is easy (an algorithm can just have both distributions explicitly hardwired into its code).

DEFINITION 6.30 (probability ensemble): *Let I be a countable index set. A* probability ensemble indexed by I *is a sequence of random variables indexed by I.*

In most cases, the set I is either the set of natural numbers $\mathbb{N}$ or an efficiently computable subset of $\{0,1\}^*$. In the case of $I = \mathbb{N}$, a probability ensemble is made up of a series of random variables $X_1, X_2, ...$, and is denoted $X = \{X_n\}_{n\in\mathbb{N}}$. The actual random variable taken in a cryptographic context is determined by the security parameter (i.e., for a given security parameter n, the random variable X_n is used). Furthermore, in most cases the random variable X_n ranges over strings of length that is polynomial in n. More specifically, it is typically the case that probability ensembles $X = \{X_n\}_{n\in\mathbb{N}}$ are considered for which there exists a polynomial $p(\cdot)$ such that for every n, the support of the random variable X_n is a subset of $\{0,1\}^{p(n)}$ (or a subset of the set of all strings of size *at most* $p(n)$). We present the formal definition for the case that $I = \mathbb{N}$.

DEFINITION 6.31 (computational indistinguishability): *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:*

$$|\Pr[D(X_n) = 1] - \Pr[D(Y_n) = 1]| \leq \mathsf{negl}(n)$$

The distinguisher D is given the unary input 1^n so that it can run in time that is polynomial in n in its attempt to distinguish. (This is of importance when the output range of X and Y may be very small.)

6.8.1 Pseudorandomness and Pseudorandom Generators

We now show that the notion of pseudorandomness is just a special case of computational indistinguishability. Then, we have the following:

DEFINITION 6.32 (pseudorandom ensembles): *An ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ is called* pseudorandom *if there exists a polynomial $\ell(n)$ such that X is computationally indistinguishable from the ensemble $U = \{U_{\ell(n)}\}_{n\in\mathbb{N}}$, where $U_{\ell(n)}$*

the uniform distribution over $\{0,1\}^{\ell(n)}$.

This can now in turn be used to redefine the notion of a pseudorandom generator, as previously defined in Definition 3.15 of Chapter 3.

DEFINITION 6.33 (pseudorandom generator): *Let* $\ell(\cdot)$ *be a polynomial and let* G *be a (deterministic) polynomial-time algorithm such that upon any input* s*, algorithm* G *outputs a string of length* $\ell(|s|)$*. 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* $\{G(s_n)\}_{n\in\mathbb{N}}$*, where* $s_n \leftarrow \{0,1\}^n$ *is pseudorandom*

Thus, pseudorandomness is just a special case of computational indistinguishability. We remark that many of the concepts that we see in this book can be cast as special cases of computational indistinguishability. Despite the fact that this involves jumping ahead (or jumping back), we give two examples. First, the decisional Diffie-Hellman (DDH) assumption of Section 7.3.2 can be formalized by stating that the ensemble of tuples of the type $(\mathbb{G}, g, g^x, g^y, g^{xy})$ is *computationally indistinguishable* from the ensemble of tuples of the type $(\mathbb{G}, g, g^x, g^y, g^z)$, where x, y, z are randomly chosen. A second example from Section 11.1.3 is that the quadratic residuosity assumption can be formalized by stating that the ensemble of quadratic residues $\mathcal{QR} = \{\mathcal{QR}_N\}$ is *computationally indistinguishable* from the ensemble of quadratic non-residues $\mathcal{QNR} = \{\mathcal{QNR}_N\}$. (Note that the index set of the above ensembles is not the set of natural numbers. In the DDH example the index set is made up of pairs $(\mathbb{G}, g)$, and in the quadratic residuosity example it is made up of values N where $N = pq$ and p and q are primes.)

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 two independent applications of G is a pseudorandom string of length $2\ell(n)$. That is, $\{(G(s_1), G(s_2))\}$ is computationally indistinguishable from the uniform distribution over $\{0,1\}^{2\ell(n)}$, where s_1 and s_2 are independently chosen random strings of length n. We prove this theorem due to its interest in its own right, and also because it is another example of a proof using the hybrid argument technique (as seen in the proof of Theorem 6.20).

We say that an ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ is efficiently samplable 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 samplable: the algorithm S chooses a random string s of length n and then outputs $G(s)$. We now prove that if two *efficiently samplable* 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. We stress that this theorem does not hold in the case that X and Y are not efficiently samplable. We will 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 unary input 1^n, but do assume that it knows the value of the security parameter and can run in time that is polynomial in n.

THEOREM 6.34 *Let X and Y be efficiently samplable ensembles such that $X \stackrel{c}{\equiv} Y$. Then, for every polynomial $p(\cdot)$, the multisample ensemble $\overline{X} = \{(X_n^{(1)}, \ldots, X_n^{(p(n))})\}_{n\in\mathbb{N}}$ is computationally indistinguishable from the multisample 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, assume by contradiction that there exists a probabilistic polynomial-time distinguisher D and a non-negligible function $\varepsilon(\cdot)$, such that:

$$\left|\Pr\left[D(X_n^{(1)}, \ldots, X_n^{(p(n))}) = 1\right] - \Pr\left[D(Y_n^{(1)}, \ldots, Y_n^{(p(n))}) = 1\right]\right| \geq \varepsilon(n)$$

For every i, 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. That is:

$$H_n^i = \left(X_n^{(1)}, \ldots, X_n^{(i)}, Y_n^{(i+1)}, \ldots, Y_n^{(p(n))}\right)$$

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. Denote $\overline{X}_n = (X_n^{(1)}, \ldots, X_n^{(p(n))})$ and likewise for

$\overline{Y}_n$. Then, we have:

$$\begin{aligned}&\left|\Pr[D(\overline{X}_n)=1]-\Pr[D(\overline{Y}_n)=1]\right| \\ &\quad=\left|\sum_{i=0}^{p(n)-1}\Pr[D(H_n^i)=1]-\sum_{i=0}^{p(n)-1}\Pr[D(H_n^{i+1})=1]\right|\end{aligned}$$

This follows from the fact that the only remaining terms in this telescopic sum are $\Pr[D(H_n^0)=1]$ and $\Pr[D(H_n^{p(n)})=1]$. By our contradicting assumption, it holds that:

$$\begin{aligned}\varepsilon(n) &\leq \left|\Pr[D(\overline{X}_n)=1]-\Pr[D(\overline{Y}_n)=1]\right| \\ &=\left|\sum_{i=0}^{p(n)-1}\Pr[D(H_n^i)=1]-\sum_{i=0}^{p(n)-1}\Pr[D(H_n^{i+1})=1]\right| \\ &\leq \sum_{i=0}^{p(n)-1}\left|\Pr[D(H_n^i)=1]-\Pr[D(H_n^{i+1})=1]\right|\end{aligned}$$

Therefore, there must exist some i for which $\left|\Pr[D(H_n^i)=1]-\Pr[D(H_n^{i+1})=1]\right|$ is non-negligible (otherwise, the entire sum would be negligible which is not the case). Notice now that the only difference between H_n^i and H_n^{i+1} is a single sample (in both, the first i samples are from X_n and the last $n-i-1$ samples are from Y_n). Thus, the fact that D can distinguish between H_n^i and H_n^{i+1} can be used to construct a distinguisher D' that can distinguish between a single sample of X and a single sample of Y, in contradiction to the assumption that $X \stackrel{\text{c}}{\equiv} Y$.

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.[7] 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,

$$\Pr[D'(X_n)=1]=\frac{1}{p(n)}\cdot\sum_{i=0}^{p(n)-1}\Pr[D(H_n^{i+1})=1]$$

[7] The efficient samplability of X and Y is needed for constructing the vector $\overline{H}_n$.

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:

$$\begin{aligned}
&|\Pr[D'(X_n) = 1] - \Pr[D'(Y_n) = 1]| \\
&= \frac{1}{p(n)} \cdot \left| \sum_{i=0}^{p(n)-1} \Pr[D(H_n^{i+1}) = 1] - \sum_{i=0}^{p(n)-1} \Pr[D(H_n^i) = 1] \right| \\
&= \frac{1}{p(n)} \cdot \left| \Pr[D(H_n^{p(n)}) = 1] - \Pr[D(H_n^0) = 1] \right| \\
&= \frac{1}{p(n)} \cdot \left| \Pr[D(\overline{X}_n) = 1] - \Pr[D(\overline{Y}_n) = 1] \right| \\
&\geq \frac{\varepsilon(n)}{p(n)}
\end{aligned}$$

which is non-negligible (a non-negligible function divided by a polynomial is always a non-negligible function). This contradicts the assumed indistinguishability of a single sample of X from Y. ■

References and Additional Reading

The notion of a one-way function was first introduced by Diffie and Hellman [50] and was later formalized and studied by Yao [134]. The concept of hard-core bits was first studied by Blum and Micali [28] and the fact that there exists a hard-core bit for every one-way function was proven by Goldreich and Levin [69]. The notion of pseudorandomness was introduced first by Yao [134] and the first construction of pseudorandom generators was given by Blum and Micali [28]. The construction of a pseudorandom generator from any one-way permutation was given by Yao [134], and the fact that pseudorandom generators can be constructed from any one-way function was shown by Håstad et al. [75]. Pseudorandom functions were defined and constructed by Goldreich, Goldwasser and Micali [67] and their extension to pseudorandom permutations was presented by Luby and Rackoff [91].

Most of the presentation in this chapter follows the textbook of Goldreich [65] (Chapters 2 and 3). 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 $f(x) = x^2$ when computed over the integers is not one-way.

6.2 Prove that if there exist one-way functions, 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 A function f is said to be length regular if for every $x, y \in \{0,1\}^*$ such that $|x| = |y|$, it holds that $|f(x)| = |f(y)|$. Show that if there exist one-way functions, then there exist length-regular one-way functions. Provide a full (formal) proof of your answer.

> **Hint:** Let f be a one-way function and let $p(\cdot)$ be a polynomial such that for every x, $|f(x)| \leq p(|x|)$ (justify the existence of this p). Define $f'(x) = f(x)10^{p(|x|)-|f(x)|}$. Prove that f' is length-regular 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 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 one-way permutations.

6.6 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.

6.7 This exercise is for students who have taken a course in complexity or are otherwise familiar with $\mathcal{NP}$ completeness.

(a) Show that the existence of one-way functions implies that $\mathcal{P} \neq \mathcal{NP}$.

(b) Assume that $\mathcal{P} \neq \mathcal{NP}$. Show that there exists a function that is easy to compute and hard to invert in the worst case, but is not one-way.

6.8 Let $x \in \{0,1\}^n$ and denote $x = x_1 \cdots x_n$. Prove that if there exist one-way functions, then there exists a one-way function f such that for every i there exists an algorithm A_i such that,

$$\Pr_{x \leftarrow U_n}[A_i(f(x)) = x_i] \geq \frac{1}{2} + \frac{1}{2n}$$

We note that $x \leftarrow U_n$ means that x is chosen according to the uniform distribution over $\{0,1\}^n$. (This exercise demonstrates that it is not possible to claim that every one-way function hides at least one *specific* bit of the input.)

6.9 Show that if a 1–1 function has a hard-core predicate, then it is one-way. Explain why the 1–1 part is needed in your proof.

6.10 Complete the proof of Proposition 6.13 by finding the Chernoff bound and applying it to the improved procedure of $\mathcal{A}'$ for guessing x_i.

6.11 Prove Claim 6.17.

6.12 Let G be a pseudorandom generator. Prove that

$$G'(x_1, \ldots, x_n) = G(x_1), G(x_2), \ldots, G(x_n)$$

where $|x_1| = \cdots = |x_n|$ is a pseudorandom generator.

Hint: Use a hybrid argument.

6.13 Prove the efficiency improvement of the generator described in the proof of Theorem 6.20. Specifically, show that $G(s) = s_{p(n)}\sigma_1, \ldots, \sigma_{p(n)-n}$ is a pseudorandom generator with expansion factor $p(n)$.

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 $\ell(n) = 4n$. Explain the connection to pseudorandom functions.

6.15 Show that if Construction 6.21 is modified so the internal nodes of the tree are also output, then the construction is no longer a pseudorandom function.

6.16 Prove that if there exist pseudorandom functions F_n that map $k(n)$ bits to one bit, then there exist pseudorandom functions that map $k(n)$ bits to n bits. Note: n denotes the security parameter, and there is no restriction on $k(\cdot)$ (in particular, k may be a constant function, or it may be $\mathsf{poly}(n)$).

Hint: Use a hybrid argument.

6.17 Prove that Construction 6.23 indeed yields a permutation, and show how the (efficient) inversion procedure works. Prove that two rounds of Construction 6.23 is *not* pseudorandom.

6.18 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 A it holds that $\{A(X_n)\}_{n\in\mathbb{N}}$ and $\{A(Y_n)\}_{n\in\mathbb{N}}$ are computationally indistinguishable.

(b) Prove that the above does not hold if 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 have seen 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 a pseudorandom permutation exists 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 have explored how pseudorandom permutations (i.e., block ciphers) are constructed in practice, and the resistance of these constructions to attack at least serves as 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. This is, all-in-all, 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 f is *one-way* if f is easy to compute but hard to invert; a formal definition is deferred to Section 7.4.1. Except briefly in Chapter 6, however, we have not yet seen any concrete examples of functions believed to be one-way.

The goal of this chapter is to introduce and discuss various problems that are believed to be "hard", and the conjectured one-way functions that can be based on these problems.[1] All the examples we explore will be *number-*

[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 mathematical problems.

theoretic in nature, and we therefore begin with a short introduction to number theory and group theory. Because we are also be interested in problems that can be solved efficiently (even a one-way function needs to be easy to compute in one direction), 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 of clarification 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 $\|N\| = \Theta(\log N)$.

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 9, 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 course:

1. As discussed above, this chapter serves as a culmination of the "top down" approach we have taken in developing private-key cryptography. That is, we have first shown that private-key cryptography can be based on pseudorandom functions and permutations, then stated (and shown in Chapter 6) that the latter can be based on one-way functions. Here, we show that one-way functions can be based on certain hard mathematical problems.

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 the following chapter.) Namely, in the private-key setting there exist suitable primitives (i.e., pseudorandom generators, functions, and permutations) for constructing schemes, and these primitives can be constructed — at least heuristically, as explored in Chapter 5 — without invoking any number theory. In the public-key setting, however, there is no single unifying primitive that suffices for constructing schemes, and we cur-

rently do not know how to construct public-key schemes "from scratch" without relying on some concrete underlying mathematical problem.

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 when a, b, 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 furthermore $a \notin \{1, b\}$ then a is also called a *non-trivial factor* of b. A positive integer $p > 1$ is *prime* if it has no non-trivial factors; i.e., it has only two divisors: 1 and p 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 with $a = qb + r$ 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 Section 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 I certainly contains some positive integers. Let d be the smallest positive integer in I. We claim 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 that $d = \gcd(a, b)$ we must show that $d \,|\, a$ and $d \,|\, b$, 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 arbitrary $c \in I$ and say $c = X'a + Y'b$ with $X', Y' \in \mathbb{Z}$. Using division with remainder (Proposition 7.1) we have $c = qd + r$ 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. So, $r = 0$ and hence $d \,|\, c$.

Since $a \in I$ and $b \in I$, the above shows that $d \,|\, a$ and $d \,|\, b$. Say there exists $d' > d$ such that $d' \,|\, a$ and $d' \,|\, b$. Then $d' \,|\, Xa + Yb$; since the latter is equal to d, this means $d' \,|\, d$. 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 Section 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 $c \,|\, ab$ and $\gcd(a, c) = 1$, then $c \,|\, b$. In particular, if p is prime and $p \,|\, ab$ then either $p \,|\, a$ or $p \,|\, b$.*

PROOF Since $c \,|\, ab$ 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 $p \not| a$ then $\gcd(a, p) = 1$. ■

PROPOSITION 7.4 *Say $p \mid N$, $q \mid N$, 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 $a \in \mathbb{Z}$ 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. 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 alternate 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, in general, 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. It is not too 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 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 $\{0, \ldots, N-1\}$.

When b is invertible modulo N we define division by b modulo N as multiplication by b^{-1} modulo N (i.e., we define $a/b = ab^{-1} \bmod N$). 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 iff* $\gcd(a, N) = 1$.

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, $ba - cN = 1$. By Proposition 7.2, this implies $\gcd(a, N) = 1$.

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 is a multiplicative inverse of a. ■

Example 7.8
Let $N = 17$ and $a = 11$. 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 done in polynomial time; see Section B.2. Exponentiation (i.e., computing $a^b \bmod N$ for $b > 0$ an integer) can also be done in polynomial time; see Section B.2.3 for further details.

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 $\circ$ such that:*

(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 additional condition 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.

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 $\mathbb{H} \neq \mathbb{G}$.

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.

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 results, we will always use multiplicative notation. *We stress that 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 will be of central importance for us.

Example 7.12

Let $N \geq 2$ be an integer. The set $\{0, \ldots, N-1\}$ with respect to addition

modulo 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 occasionally use $\mathbb{Z}_N$ also to denote the set $\{0, \ldots, N-1\}$ without regard to any particular 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 excruciating detail:

$$a = a \cdot 1 = a(c \cdot c^{-1}) = (ac)c^{-1} = (bc)c^{-1} = b(c \cdot c^{-1}) = b \cdot 1 = b.$$

■

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 to the case when m is zero or a negative integer in the natural way. (In general, we leave g^r undefined if r 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 non-trivial observation is discussed in Section 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. 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 so $e^{-1} \bmod m$ exists. 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 third-to-last equality follows from Corollary 7.15. ■

7.1.4 The Group $\mathbb{Z}_N^*$ and the Chinese Remainder Theorem

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? Clearly, '1' will have to be the identity. We know that not every element in this set is invertible;

for example, '0' 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 $a \in \{0, \ldots, N-1\}$ are invertible modulo N? Proposition 7.7 says that this occurs if and only if $\gcd(a, N) = 1$. We have also seen in Section 7.1.2 that whenever a is invertible, it has an inverse lying in the range $\{0, \ldots, N-1\}$. 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) with the associated binary operation of multiplication modulo N.

We claim that $\mathbb{Z}_N^*$, with respect to this operation, is a group. The discussion above shows that $\mathbb{Z}_N^*$ has an identity, and that each element in this set 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 $N = pq$, 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 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 $|\mathbb{Z}_N^*| = 8 = 4 \cdot 2 = \phi(15)$. The inverse of 8 in $\mathbb{Z}_N^*$ is 2, since $8 \cdot 2 = 16 = 1 \bmod 15$. ♢

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 when $N = p$ 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.*

Group Isomorphisms

An *isomorphism* of a group $\mathbb{G}$ provides an alternate, 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; (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 these groups are* isomorphic *and write this as $\mathbb{G} \simeq \mathbb{H}$.*

Note that if $\mathbb{G}$ is finite and $\mathbb{G} \simeq \mathbb{H}$, then $\mathbb{H}$ must be finite and have 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 $\mathbb{G}$.

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 can 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 nn' 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 7.5 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

$$f(x) = ([x \bmod p],\, [x \bmod q]).$$

Then f is an isomorphism from $\mathbb{Z}_N$ to $\mathbb{Z}_p \times \mathbb{Z}_q$ as well as[2] 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 (say) $x_p \notin \mathbb{Z}_p^*$ 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 $x \in \mathbb{Z}_N^*$.

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 $+_N, +_p, +_q$ denote addition modulo N, modulo p, and modulo q, respectively; 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

$$\begin{aligned} f(a +_N b) &= \Big([(a +_N b) \bmod p],\, [(a +_N b) \bmod q]\Big) \\ &= \Big([(a +_p b) \bmod p],\, [(a +_q b) \bmod q]\Big) \\ &= \Big([a \bmod p],\, [a \bmod q]\Big) \boxplus \Big([b \bmod p],\, [b \bmod q]\Big) = f(a) \boxplus f(b). \end{aligned}$$

[2] Technically, here we consider the *restriction* of f to inputs in $\mathbb{Z}_N^*$.

(In Exercise 7.6, you are asked to prove that $(a +_N b) \bmod p = (a +_p b) \bmod p$, a result used for the second equality, above.) ■

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 $p \stackrel{\text{def}}{=} \prod_{i=1}^{\ell} p_i$, then

$$\mathbb{Z}_p \simeq \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_\ell} \quad \text{and} \quad \mathbb{Z}_p^* \simeq \mathbb{Z}_{p_1}^* \times \cdots \times \mathbb{Z}_{p_\ell}^*.$$

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. ♢

7.1.5 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 will 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 $14 \cdot 13 \bmod 15$. 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 $11^2 \bmod 15$. Exercise 7.25 gives $11 \leftrightarrow (1,2)$ and so $[11^2 \bmod 15] \leftrightarrow (1,2)^2$ (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 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. Section 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. Section 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. It can similarly 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 pre-processing 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 $\{0, \ldots, 34\}$. 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 $29 \leftrightarrow ([29 \bmod 5], [29 \bmod 7]) = (-1, 1)$. 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 $1 = [29^{100} \bmod 35]$. ♢

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" 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 *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 the advent of 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 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 correctly solves the factoring problem has been developed despite much 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, each 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 certain contexts, we would like to prevent this.

As this discussion indicates, the "hardest" numbers to factor are 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 Section B.2.4 for choosing random integers in a certain range can be used to generate random n-bit primes. (The discussion in Section 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. That is:

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

(An alternate to outputting fail in the last line is simply to output an arbitrary n-bit integer, which may not be prime.) 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'.

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 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.

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 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).[3]

When using a randomized primality test of this sort in the prime-generation algorithm shown earlier, the output of the algorithm is still a random prime of the desired length conditioned on the output being prime. An additional source of error (besides the possibility of outputting fail) is introduced, however, 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.

[3]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.

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 the remaining probability).*

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.)

ALGORITHM 7.34
Generating a random prime

Input: Length 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

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 called *strong primes*). In this case, appropriate modifications of the prime-generation algorithm shown above can be used. 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 $a^{N-1} \stackrel{?}{=} 1 \bmod N$. (Recall that exponentiation and computation of greatest common divisors can be done in polynomial time. Choosing a random element in the range $\{1, \ldots, N-1\}$ can also be done in polynomial time. Refer to Section B.2.) 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. See Algorithm 7.35

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"

If N is prime then the discussion above implies that this algorithm always outputs "prime." If N is composite, the algorithm outputs "composite" if it finds (in any of the t iterations) an $a \in \mathbb{Z}_N^*$ such that $a^{N-1} \neq 1 \bmod N$. (It also outputs "composite" if it ever finds an $a \notin \mathbb{Z}_N^*$; we will take this into account later.) We will refer to an a 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, so that in each iteration the algorithms finds such a witness with "high" probability. This intuition is correct *provided there is at least one witness that N is composite*.

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 an 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 $\bar{\mathbb{H}} \stackrel{\text{def}}{=} \{\bar{h}h \mid h \in \mathbb{H}\}$ (this is not a subgroup of $\mathbb{G}$). 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.

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). ■

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

$$\frac{\frac{|\mathbb{Z}_N^*|}{2} + \left((N-1) - |\mathbb{Z}_N^*|\right)}{N-1} = 1 - \frac{|\mathbb{Z}_N^*|/2}{(N-1)} \geq 1 - \frac{|\mathbb{Z}_N^*|/2}{|\mathbb{Z}_N^*|} = \frac{1}{2},$$

and so the probability that the algorithm does not find a witness in any of the t iterations (and hence 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 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 values $a^u, a^{2u}, \ldots, a^{2^r u}$ (all modulo N). Note that 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 later):

LEMMA 7.39 *Say x 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 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 $N \mid (x+1)$ or $N \mid (x-1)$ by Proposition 7.3. However, these can only possibly occur if $x = [\pm 1 \bmod N]$. ■

Now, say N is prime and fix arbitrary $a \in \mathbb{Z}_N^*$. Let i be the minimum (non-negative) 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 prime, then 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 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 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 $\overline{\mathsf{Bad}}$ and show that (1) Bad is a subset of $\overline{\mathsf{Bad}}$, and (2) $\overline{\mathsf{Bad}}$ is a strict subgroup of $\mathbb{Z}_N^*$. Together, these imply that $|\mathsf{Bad}| \leq |\mathbb{Z}_N^*|/2$ as in Theorem 7.38, and so at least half the elements of $\mathbb{Z}_N^*$ are strong witnesses. (We stress that we do not claim that Bad is a subgroup.)

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 $a^{2^i u} = -1 \bmod N$; alternately, 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.

Define

$$\overline{\mathsf{Bad}} \stackrel{\text{def}}{=} \{a \mid a^{2^i u} = \pm 1 \bmod N\}.$$

We now prove what we claimed above.

CLAIM 7.41 $\mathsf{Bad} \subseteq \overline{\mathsf{Bad}}$.

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 $a \in \overline{\mathsf{Bad}}$. In the second case, we have $j \leq i$ by choice of i. If $j = i$ then clearly $a \in \overline{\mathsf{Bad}}$. If $j < i$ then $a^{2^i u} = (a^{2^j u})^{2^{i-j}} = 1 \bmod N$ and $a \in \overline{\mathsf{Bad}}$. Since a was arbitrary, this shows $\mathsf{Bad} \subseteq \overline{\mathsf{Bad}}$.

CLAIM 7.42 $\overline{\mathsf{Bad}}$ *is a subgroup of* $\mathbb{Z}_N^*$.

Clearly, $1 \in \overline{\mathsf{Bad}}$. Furthermore, if $a, b \in \overline{\mathsf{Bad}}$ then

$$(ab)^{2^i u} = a^{2^i u} b^{2^i u} = (\pm 1)(\pm 1) = \pm 1 \bmod N.$$

By Lemma 7.36, $\overline{\mathsf{Bad}}$ is a subgroup.

CLAIM 7.43 $\overline{\mathsf{Bad}}$ *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 $a \in \overline{\mathsf{Bad}}$ 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 that corresponds to $(a_1, 1)$. 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 $b \notin \overline{\mathsf{Bad}}$. ■

An integer N is a *perfect power* if $N = \hat{N}^e$ for some integers $\hat{N}$ and $e > 1$ (here it is not required for $\hat{N}$ to be prime). Note that any prime power is also a perfect power. We can now describe a primality testing algorithm in full.

Exercises 7.9 and 7.10 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 easily complete the proof of Theorem 7.33.

ALGORITHM 7.44
The Miller-Rabin primality test

Input: Integer N 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"

PROOF If N is prime, then there are no strong witnesses for N and so the Miller-Rabin algorithm always outputs "prime". If N is composite there are two cases: if N is a prime power then 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

$$\frac{|\mathbb{Z}_N^*|/2 + \Big((N-1) - |\mathbb{Z}_N^*|\Big)}{N-1} = 1 - \frac{|\mathbb{Z}_N^*|/2}{N-1} \geq 1 - \frac{|\mathbb{Z}_N^*|/2}{|\mathbb{Z}_N^*|} = \frac{1}{2},$$

and so the probability that the algorithm does not find a witness in any of the t iterations (and hence outputs "prime") is at most 2^{-t}. ■

7.2.3 The Factoring Assumption

Now that we have shown how to generate random primes, we formally define the factoring 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. (A natural way to construct such an algorithm is to generate two random primes p and q of length n, as discussed in Section 7.2.1, and then set N to be their product.) 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.

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 (as we will see in Section 7.4.1), the factoring assumption *in the form we have described it* does not seem very useful for practical cryptographic constructions.[4] 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 known as 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) and so "working in the exponent modulo $\phi(N)$" is not an available option, at least not in the 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 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

[4] In Section 11.2.2, however, we will see a very useful problem whose hardness can be shown to be *equivalent* to that of factoring.

$e > 0$ with $\gcd(e, \phi(N)) = 1$ and an integer d satisfying $ed = 1 \bmod \phi(N)$. (Note that 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* 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 an 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 different 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? Note that 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 stress that this is a *necessary*, but not a *sufficient*, condition.) 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. Another way of saying this is that 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 7.11) 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 (see Exercise 7.12 for a simple case of this result). It is possible, however, that there are other ways of solving the RSA problem that do not involve explicit computation of $\phi(N)$ or d, and so we cannot conclude that the RSA problem is as hard as factoring.

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 too hard to verify that $\langle g \rangle$ is a subgroup of $\mathbb{G}$ for any g (see Exercise 7.3); $\langle g \rangle$ is called 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 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 will 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 our scope, 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 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 $0 < i' < 15$.

$\mathbb{Z}_{15}$ 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 $\{0, 10, 20\}$. Note that 5 and 3 both divide $|\mathbb{Z}_{15}| = 15$ as required by Proposition 7.51. ♢

Example 7.55
Consider the 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 gener-

ator. Indeed, for any element $g \in \{1, \ldots, p-1\}$ and integer $i > 0$ we have $ig = 0 \bmod p$ iff $p \,|\, ig$. But then Proposition 7.3 says that either $p \,|\, g$ or $p \,|\, i$. The former cannot occur (since $g < p$), and the smallest positive integer for which the latter can occur is $i = p$. We have thus shown that every non-zero element g 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 shows that all cyclic groups of the same order are, in some sense, the same.

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. ♢

We stress that while the above is true in a *group-theoretic* sense, it is not true in a *computational* sense. That is, although (for example) $\mathbb{Z}_p^*$, 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.

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 $x = \log_g h$.[5] Note that if $g^{x'} = h$ for some arbitrary integer x', then $\log_g h = [x' \bmod q]$.

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 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 output* $(\mathbb{G}, q, g)$*, where* $\mathbb{G}$ *is a cyclic group of order* q *(with* $\|q\| = n$*) and* g *is a generator of* $\mathbb{G}$*.*
2. *Choose* $h \leftarrow \mathbb{G}$*. (Note that 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* 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.

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 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 $h_1 = g^{x_1}$ and $h_2 = g^{x_2}$ then

$$\mathsf{DH}_g(h_1, h_2) = g^{x_1 \cdot x_2} = h_1^{x_2} = h_2^{x_1}.$$

[5] 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.

The *CDH problem* is to compute $\mathsf{DH}_g(h_1, h_2)$ given randomly-chosen h_1 and h_2.

Note that if the discrete logarithm problem in $\mathbb{G}$ is easy, then the CDH problem is too: given h_1 and h_2, first compute $x_1 = \log_g h_1$ and then output the answer $h_2^{x_1}$. In contrast, it is not clear whether hardness of the discrete logarithm problem implies that the CDH problem is necessarily hard as well.

The *DDH problem*, roughly speaking, is to distinguish $\mathsf{DH}_g(h_1, h_2)$ from a random group element; that is, given randomly-chosen h_1, h_2 and a candidate solution y, to decide whether $y = \mathsf{DH}_g(h_1, h_2)$ or whether y was chosen at random from $\mathbb{G}$. Formally, let $\mathcal{G}$ be as above; that is, $\mathcal{G}$ is 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, where the group operation in $\mathbb{G}$ can be computed efficiently (namely, in time polynomial in n). Then:

DEFINITION 7.60 *We say* 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 for some group $\mathbb{G}$, then the CDH problem is too. Similarly, if the CDH problem is easy in $\mathbb{G}$ then so is the DDH problem; you are asked to show this in Exercises 7.13 and 7.14. 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.

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 widely 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* shown 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*. 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 Section B.3), using a prime-order group can thus potentially yield a more efficient algorithm $\mathcal{G}$ (which, recall, needs to compute a generator g of the group $\mathbb{G}$ that it outputs).

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.[6] We show that when the group order q is prime, this is (almost) true.

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 $\mathbb{Z}_q$,

$$\begin{aligned}\Pr[\mathsf{DH}_g(g^{x_1}, g^{x_2}) = 1] &= \Pr[x_1 = 0 \text{ or } x_2 = 0] \\ &= 1 - \Pr[x_1 \neq 0] \cdot \Pr[x_2 \neq 0] = 1 - \left(1 - \frac{1}{q}\right)^2.\end{aligned}$$

[6] 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.

Fix any $y \in \mathbb{G}$, $y \neq 1$, and let $x = \log_g y \neq 0$. Note that $\mathsf{DH}_g(g^{x_1}, g^{x_2}) = y$ iff $x_1 x_2 = x \bmod q$. Since q is prime, all non-zero elements of $\mathbb{Z}_q$ have a multiplicative inverse modulo q, and so $x_1 x_2 = x \bmod q$ iff x_2 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. ∎

A uniformly-distributed element y' has $\Pr[y' = y] = \frac{1}{q}$ 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 uniformly distributed 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. Instead, this should merely be viewed as a heuristic reason why prime-order groups are often used.

7.3.3 Working in (Subgroups of) $\mathbb{Z}_p^*$

Groups of the form $\mathbb{Z}_p^*$, for p prime, 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 Section 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 $\mathcal{G}_1$.

Note, however, that the cyclic group $\mathbb{Z}_p^*$ (for $p > 2$ 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 11.8 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.

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. This class of groups is very useful for cryptography since the DDH problem is assumed to be hard for such groups.

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 prime p
  with $q \stackrel{\text{def}}{=} (p-1)/2$ also prime
choose arbitrary $x \in \mathbb{Z}_p^*$ with $x \neq \pm 1 \bmod p$
$g := x^2 \bmod p$
return p, q, g

Generation of p can be done 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 widely 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 elliptic-curve groups. (See Chapter 8 for further discussion.) Although elliptic-curve groups are very 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. In any case, 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; 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.

Let $p \geq 5$ be a prime, and let $\mathbb{Z}_p \stackrel{\text{def}}{=} \{0, \ldots, p-1\}$.[7] 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 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.)

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; a value of x for which $f(x) = 0 \bmod 7$ gives one point on the curve. This allows us to determine the points on the curve:

[7] 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 for fields of characteristic not equal to 2 or 3.

- $f(0) = 3 \bmod 7$, a quadratic non-residue modulo 7.
- $f(1) = 0 \bmod 7$, so we obtain the point $(0, 0) \in E(\mathbb{Z}_7)$.
- $f(2) = 3 \bmod 7$, 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 $E(\mathbb{Z}_7)$. $\diamondsuit$

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 **??**. We stress that 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 is possible to show 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[8] a binary operation, called 'addition' and denoted by '+,' on points of $E(\mathbb{Z}_p)$ in the following way:

- 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.)

[8] Our approach is informal, and in particular we do not justify that it leads to a consistent definition.

Rules for negation and addition of arbitrary points are consequences of these rules.

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, e.g., by looking at Figure **??**, this means that $-P$ is simply the reflection of P in the x-axis; that is, if $P = (x, y)$ then $-P = -(x, y) = (x, -y)$.

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; note that it may be that $P_3 = \mathcal{O}$. 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 this does not involve a division by 0. The line passing through P_1 and P_2 has the equation

$$y = m \cdot (x - x_1) + y_1 \bmod p\,.$$

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

$$x_3 = [m^2 - x_1 - x_2 \bmod p\,] \qquad\qquad y_3 = [m \cdot (x_3 - x_1) + y_1 \bmod p\,].$$

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:

CLAIM 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*

$$x_3 = [m^2 - x_1 - x_2 \bmod p] \qquad y_3 = [m \cdot (x_1 - x_3) - y_1 \bmod p],$$

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 claim.

CLAIM 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*

   $$x_3 = [m^2 - 2x_1 \bmod p] \qquad y_3 = [m \cdot (x_1 - x_3) - y_1 \bmod p],$$

   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 a prime-order, and hence cyclic, group. Efficient methods for doing so are beyond the scope of this book.

7.4 Applications of Number-Theoretic Assumptions in Cryptography

We have by now 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 be shown in 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 note that their existence follows from all the number-theoretic hardness assumptions we have seen in this chapter.

Informally, a function f is said to be *one-way* if f is easy to compute but f^{-1} is not. Formally (the following is just a re-statement of Definition 6.1):

DEFINITION 7.66 *A function $f : \{0,1\}^* \to \{0,1\}^*$ whose output length is polynomially-related to its input length*[9] *is* one-way *if the following two conditions hold:*

1. Easy to compute: *There exists a polynomial-time algorithm that on input $x \in \{0,1\}^*$ outputs $f(x)$.*

2. Hard to invert: *Consider the following experiment for a given algorithm $\mathcal{A}$ and parameter n:*

   The inverting experiment $\mathsf{Invert}_{\mathcal{A},f}(n)$

   (a) Choose input $x \leftarrow \{0,1\}^n$. Compute $y := f(x)$.
   (b) $\mathcal{A}$ is given y as input, and outputs x'.
   (c) The output of the experiment is defined to be 1 if $f(x') = y$, and 0 otherwise.

   Then it is required that for all probabilistic, polynomial-time algorithms $\mathcal{A}$ there exists a negligible function negl *such that*

$$\Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1] \leq \mathsf{negl}(n).$$

[9] Recall this means that there exists a constant $c > 0$ such that $|f(x)| \geq |x|^c$.

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 $\mathsf{GenModulus}$ here 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. We define a *deterministic* function f_{Gen} (that can be computed in polynomial time) by describing a deterministic polynomial-time algorithm for computing f_{Gen}. This algorithm runs Gen as a subroutine, but we stress that f_{Gen} is *deterministic* since the random tape of Gen is fixed to the input x of f_{Gen}, rather than being chosen at random each time $f_{\mathsf{Gen}}(x)$ is computed.

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, note that 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 at random.

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 nontrivial factor of N (by running $\mathsf{Gen}(1^n; x)$ to obtain (N, p, q) and outputting the factors p and q). We thus 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). In fact, hardness of the RSA problem gives something stronger: a family of one-way *permutations*. We define this primitive here, but once again refer the reader to Chapter 6 for a more in-depth discussion of its application.

The following is just a re-statement of Definitions 6.2 and 6.3:

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 we define next.*

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 additionally holds that* $\mathcal{D}_I = \mathcal{R}_I$*, and the function* $f_I : \mathcal{D}_I \to \mathcal{D}_I$ *is a bijection.*

Note that, 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 uniformly-distributed value of y.

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 *Let* $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ *be a family of functions.* Π *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 GenModulus 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 (and, by extension, the CDH or DDH problems) in $\mathbb{Z}_p^*$, with p prime, implies the existence of a one-way permutation family. See Exercise 7.17.

CONSTRUCTION 7.71

- *Parameter-generation algorithm* Gen: on input 1^n, run $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and output $I = \langle N, e \rangle$. We will have $\mathcal{D}_I = \mathbb{Z}_N^*$.
- *Sampling algorithm* Samp: on input $I = \langle N, e \rangle$, choose a random element of $\mathbb{Z}_N^*$.
- *Evaluation algorithm* f: on input $I = \langle N, e \rangle$ and $x \in \mathbb{Z}_N^*$, outputs $[x^e \bmod N]$.

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 provable constructions of such hash functions based on more primitive 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 hardness of factoring is described in Exercise 7.18. 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 also 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 anyway from 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 as Construction 7.72.

CONSTRUCTION 7.72

Let $\mathcal{G}$ be as described in the text. Define (Gen, H) as follows:

- *Key generation algorithm* Gen: On input 1^n, run $\mathcal{G}(1^n)$ to obtain $(\mathbb{G}, q, g)$ and then select $h \leftarrow \mathbb{G}$. Output $s = \langle \mathbb{G}, q, g, h \rangle$.
- *Hash algorithm* H: On input $s = \langle \mathbb{G}, q, g, h \rangle$ and message $(x_1, x_2) \in \mathbb{Z}_q \times \mathbb{Z}_q$, output $g^{x_1} h^{x_2} \in \mathbb{G}$.

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 inout 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 an element of $\mathbb{G}$, but we can view the output of H_s as a bit-string if we fix some representation of $\mathbb{G}$. To satisfy the requirements of Definition 4.10 (which requires the output length to be fixed as a function of n) we can pad the output with 0s 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 we show in Exercise 7.74, however, compression can be achieved when using subgroups of $\mathbb{Z}_p^*$ of the type discussed in Section 7.3.3.

THEOREM 7.73 *If the discrete logarithm problem is hard relative to $\mathcal{G}$, then Construction 7.72 is collision resistant.*

PROOF Let $\Pi = (\mathsf{Gen}, H)$, and let $\mathcal{A}$ be an arbitrary probabilistic, polynomial-time algorithm with

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n) = 1]$$

(cf. Definition 4.10). Consider the following algorithm $\mathcal{A}'$ solving the discrete logarithm problem relative to $\mathcal{G}$:

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 $[\Delta \bmod q] \neq 0$ 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 there was a collision. Since q is prime and $\Delta \neq 0 \bmod q$, the inverse Δ^{-1} exists. Raising each side of Equation (7.3) to the power Δ^{-1} gives:

$$g^{(x_1 - x'_1) \cdot \Delta^{-1}} = \left(h^{x'_2 - x_2}\right)^{\Delta^{-1}} = h^{[\Delta \cdot \Delta^{-1} \bmod q]} = h^1 = h,$$

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. ■

The above only shows collision resistance, but does not necessarily mean that Construction 7.72 is compressing. Indeed, as discussed earlier, whether or not the construction is compressing depends on the number of bits required to represent elements of $\mathbb{G}$. For many natural choices of groups, however, compression is attained; you are asked to prove this for one concrete example in Exercise 7.16. Interestingly, 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.

Example 7.74

Let $p = 47$, and say we work with the subgroup of quadratic residues in $\mathbb{Z}_{47}^*$ having order $q = (47-1)/2 = 23$. (See Section 7.3.3.) Since $|q| = 5$, any 4-bit binary string can be viewed as an element of $\mathbb{Z}_q = \{0, \ldots, q-1\}$ in the natural way. Similarly, any element of $\mathbb{Z}_p^* = \{1, \ldots, 46\}$ can be written in the natural way using exactly 6 bits (padding to the left with zeros as needed).

Say $g = 4$ and $h = 21$, so that $s = \langle \mathbb{G}, 23, 4, 21 \rangle$. We obtain a function H_s mapping 8-bit inputs to 6-bit outputs. For example, $H_s(11111101)$ is computed as follows: 1111 in binary is the number 15; 1101 is the number 13. Then $4^{15} \cdot 21^{13} = 3 \bmod 47$, which can be written in binary as 000011. So, $H_s(11111101) = 000011$. ♢

References and Additional Reading

The book by Childs [36] has excellent coverage of the topics discussed in this chapter (and more), in greater depth but at a similar level of exposition. Shoup [117] 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 [58] and Herstein [77]; 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 [121]. The Miller-Rabin test is due to Miller [97] and Rabin [108]. A deterministic primality test was recently discovered by Agrawal et al. [11]. See Dietzfelbinger [49] for an accessible and comprehensive presentation of primality testing.

The RSA permutation was introduced by Rivest, Shamir and Adleman [110] and has since been studied greatly. The discrete logarithm and Diffie-Hellman problems were first considered, at least implicitly, by Diffie and Hellman [50]. Recent surveys of each of these problems and their applications are given by Odlyzko [101] and Boneh [29].

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 [118] is perhaps an exception. As is the case for 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 [130], 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. [73].

The construction of a collision-resistant hash function based on the discrete logarithm problem is due to [35], and an earlier construction based on the

hardness of factoring is given in [71] (see Exercise 7.18).

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 Prove that if $\mathbb{G}, \mathbb{H}$ are groups, then $\mathbb{G} \times \mathbb{H}$ is a group.

7.6 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 $[X \bmod N]$.

7.7 Fill in the details of the proof of the Chinese remainder theorem, showing that $\mathbb{Z}_N^*$ is isomorphic to $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.

7.8 Corollary 7.21 shows that if $N = pq$ and $ed = 1 \bmod \phi(N)$ then for any $x \in \mathbb{Z}_N^*$ 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.

7.9 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 $\hat{N}$ such that $\hat{N}^e = N$. (*Hint*: use binary search.)

(c) Given N, show how to test in $\mathsf{poly}(\|N\|)$ time whether N is a perfect power.

7.10 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.

7.11 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.

7.12 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.

7.13 Prove formally that the hardness of the CDH problem relative to $\mathcal{G}$ implies the hardness of the discrete logarithm problem relative to $\mathcal{G}$.

7.14 Prove formally that the hardness of the DDH problem relative to $\mathcal{G}$ implies the hardness of the CDH problem relative to $\mathcal{G}$.

7.15 Prove the third statement in Claim 7.65.

7.16 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.

7.17 Let $\mathcal{G}_1$ be as in Section 7.3.3. Show that the 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 $\mathbb{Z}_p^*$.

7.18 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 collision resistant.

Hint: Show that the eth root of y can be computed from any collision.

CONSTRUCTION 7.75

Define (Gen, H) as follows:

- $\mathsf{Gen}(1^n)$ runs $\mathsf{GenRSA}(1^n)$ to obtain N, e, and selects $y \leftarrow \mathbb{Z}_N^*$. The seed is $s = \langle N, e, y \rangle$.
- If $s = \langle N, e, y \rangle$, then H_s maps inputs in $\{0,1\}^{3n}$ to outputs in $\mathbb{Z}_N^*$. Let $f_s^0(x) \stackrel{\text{def}}{=} [x^e \bmod N]$ and $f_s^1(x) \stackrel{\text{def}}{=} [y \cdot x^e \bmod N]$. For a $3n$-bit long string $X = X_1 \cdots X_{3n}$, define

$$H_s(X) \stackrel{\text{def}}{=} f_s^{X_1}\left(f_s^{X_2}\left(\cdots\left(1\right)\cdots\right)\right).$$

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 $\mathbb{Z}_p^*$. But this does not mean that brute-force search is the best available method for attacking these problems! Here, we survey some better algorithms for these problems. Besides being interesting in their own right, and serving as a nice application of some of the number theory we have already learned, 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 5 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) 5 years to successfully factor N.

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 when 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 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 some notation developed in Sections 7.1.4 and 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 one-to-one mapping 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 $\mathcal{O}(\sqrt{N})$. 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 *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 asymptotic running time) is the *general number field sieve*. Heuristically,[2] 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 $\mathcal{O}(n)$.

8.1.1 Pollard's $p-1$ Method

This section relies on some of the material from Section B.3.1.

An algorithm due to Pollard can be used to factor an integer $N = pq$ when $p-1$ has only "small" factors. The key to 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 Section 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 had an integer B for which

$$(p-1) \mid B \quad \text{but} \quad (q-1) \nmid B. \tag{8.2}$$

[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.

(We defer until later the details of how such a B is determined.) Write $B = \gamma \cdot (p-1)$ for some integer γ. Taking an arbitrary element $x \in \mathbb{Z}_N^*$ and setting $y = [x^B \bmod N]$ (note that y can be computed using the efficient exponentiation algorithm from Section B.2.3), 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 $p-1$. The value y with $y \leftrightarrow (1, x_q^B)$ therefore satisfies Equation (8.1) as long as $x_q^B \neq 1 \bmod q$. We now show that the latter occurs with sufficiently high probability.

Since $\mathbb{Z}_q^*$ is cyclic, the group $\mathbb{Z}_q^*$ contains exactly $\phi(q-1)$ generators each of whose order is (by definition) $q-1$. 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

$$x_q^B = x_q^{\alpha \cdot (q-1) + \beta} = x_q^\beta \bmod q,$$

using Theorem 7.14. But $x_q^\beta \neq 1 \bmod q$ since the order of x_q is strictly larger than β.

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 $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.) Using Theorem B.16, we conclude that with probability at least $\frac{\phi(q-1)}{q-1} = \Omega(1/\log q)$ we choose x such that $x_q^B \neq 1 \bmod q$.

Pseudocode for Pollard's $p-1$ algorithm follows. The discussion in the previous paragraphs implies that the 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.

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

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 $(q-1) \mid B$. It is, of course, possible to run the algorithm repeatedly using multiple choices for k. Other ways of selecting B are also possible.

Pollard's $p-1$ method is thwarted if both $p-1$ and $q-1$ have large prime factors. 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; 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? Say we choose values $x_1, \ldots, x_k$ independently and uniformly at random from $\mathbb{Z}_N^*$, where $k = 2^{n/2} = \mathcal{O}(\sqrt{p})$. Note that:

- By a direct application of Lemma A.9, 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.

- 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 $[x_i \bmod p] = [x_j \bmod p]$ 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 $\mathcal{O}(\sqrt{p})$ 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. (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 explained in the following claim.

CLAIM 8.2 *Let $x_1, \ldots$ be a sequence of values with $x_m = F(x_{m-1}) \bmod N$. Say $x_i = x_j \bmod p$ with $i < j \leq k$. Then there exists an $i' < j \leq k$ 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 $j - i$. (To see this, observe that $x_m = F(x_{m-1}) \bmod p$ for all m. So $x_{i+\delta} = x_{j+\delta} \bmod p$ for all $\delta \geq 0$ and then $x_i = x_j = x_{i+(j-i)} = x_{j+(j-i)} \bmod p$.) Take i' to be the least 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 $x_{i'} = x_{2i'} \bmod p$. ∎

(The proof of the claim is the reason the algorithm is called "rho," since the repeating sequence suggests the Greek letter ρ as shown in Figure **??**.)

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
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 practuce. (See [127, 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 around 300 bits long. In this section, we describe the general principles underlying the quadratic sieve algorithm but caution the reader that many of the 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 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 in polynomial time a non-trivial factor of N. 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 $N \mid (x-y)(x+y)$. On the other hand, $N \nmid (x-y)$ 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 11.20.

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[3] two-step process:

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 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 $x_i > \sqrt{n}$, so $x_i^2 > n$ 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:

$$\begin{aligned} x_1^2 &= \prod_{i=1}^{k} p_i^{e_{1,i}} \bmod N \\ &\vdots \\ x_\ell^2 &= \prod_{i=1}^{k} p_i^{e_{\ell,i}} \bmod N. \end{aligned} \tag{8.3}$$

Looking only at the exponents of each p_i modulo 2, we obtain the matrix

$$\Gamma \stackrel{\text{def}}{=} \begin{pmatrix} \gamma_{1,1} & \gamma_{1,2} & \cdots & \gamma_{1,k} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{\ell,1} & \gamma_{\ell,2} & \cdots & \gamma_{\ell,k} \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} e_{1,1} \bmod 2 & e_{1,2} \bmod 2 & \cdots & e_{1,k} \bmod 2 \\ \vdots & \vdots & \ddots & \vdots \\ e_{\ell,1} \bmod 2 & e_{\ell,2} \bmod 2 & \cdots & e_{\ell,k} \bmod 2 \end{pmatrix}.$$

[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.

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 $2^{\mathcal{O}(\sqrt{n \cdot \log n})}$ 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 Section 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 $g^x = y$.[4] 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 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.

[4] 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.

We next look at computing discrete logarithms in some specific groups. As an illustrative but simple example, we first 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 exercise is to demonstrate that

> *even though any cyclic group of order q is isomorphic to $\mathbb{Z}_q$ (cf. Example 7.58 in Chapter 7), and hence all cyclic groups of the same order are, in some sense, "the same", the hardness of the discrete logarithm problem depends in a crucial way on the particular* **representation** *being used for the group.*

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, 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.

Currently, the best-known algorithm for computing discrete logarithms in $\mathbb{Z}_p^*$ (for p prime) is the *general number field sieve*.[5] 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 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 512-bit prime p, the general number field sieve computes discrete

[5] It is no accident that the name of this algorithm and its running time are the same as for that of the currently best-known algorithm for factoring: they share many of the same underlying steps.

logarithms in $\mathbb{Z}_p^*$ in roughly $2^{512^{1/3} \cdot 9^{2/3}} \approx 2^{8 \cdot 4} = 2^{32}$ 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 64-bit prime, since then $\sqrt{q} \approx 2^{64/2} = 2^{32}$. 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 players, but that are equally hard for an adversary to break.

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 as 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 at least $\Omega(q)$ time. Instead, we "mark 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 $\log_g y = [kt - i \bmod q]$. Psuedocode for this algorithm is given next.

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 $g_i = g_{i-1} \cdot g_1$.) 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

$$2^0 = 1, \;\; 2^5 = 3, \;\; 2^{10} = 9, \;\; 2^{15} = 27, \;\; 2^{20} = 23, \;\; 2^{25} = 11.$$

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** $[kt - i \bmod q]$

(We omit the "mod 29" since it is understood that operations are in the group $\mathbb{Z}_{29}^*$.) 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 $i \geq q/p$. 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 are interested in finding 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

$$\left(g^{q/q_i}\right)^x = (g^x)^{q/q_i} = y^{q/q_i} \text{ for } i = 1, \ldots, k. \tag{8.4}$$

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. (This is of course the best we can hope for, since the equation $g^x = y$ only uniquely determines x modulo q.) The answer x itself can be efficiently reconstructed from $x_1, \ldots, x_k$.

Example 8.8

We again apply the ideas introduced here to 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

$$\begin{aligned} (g^x)^{30/5} = y^{30/5} &\Rightarrow (3^6)^x = 16^x = 26^6 = 1 \\ (g^x)^{30/3} = y^{30/3} &\Rightarrow (3^{10})^x = 25^x = 26^{10} = 5 \\ (g^x)^{30/2} = y^{30/2} &\Rightarrow (3^{15})^x = 30^x = 26^{15} = 30. \end{aligned}$$

(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 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 8.2.

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 8.3 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 will discuss the main ideas used by the index calculus method but leave the details beyond the scope of our treatment. Also, some small changes are made in order to simplify the presentation.

The index calculus method uses a two-stage process. Importantly, the first stage requires knowledge only of the modulus p and the base g and so it can be run as a 'pre-processing step' before y is known. For the same reason, it suffices to run the first stage 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 stage, 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, 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 $\{e_{i,j}\}$ 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

$$g^{x^*} \cdot y = \prod_{i=1}^{k} p_i^{e_i^*} \bmod p$$
$$\Rightarrow x^* + \log_g y = \sum_{i=1}^{k} e_i^* \cdot \log_g p_i \mod (p-1),$$

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[6] 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 $\log_g y$.

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 $3^{14} = 13 \bmod 101$. That is,

$$\begin{aligned} 10 &= \log_3 5 + \log_3 13 \mod 100 \\ 12 &= 4 \cdot \log_3 2 + \log_3 5 \mod 100 \\ 14 &= \log_3 13 \mod 100. \end{aligned}$$

We also have $3^5 \cdot 87 = 32 = 2^5 \bmod 101$, or

$$5 + \log_3 87 = 5 \cdot \log_3 2 \mod 100. \tag{8.6}$$

Using simple algebraic manipulation, we first 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 8.3). 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 for the quadratic sieve method.

[6] 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.

References and Additional Reading

The texts by Wagstaff [127], Shoup [117], Crandall and Pomerance [43], and Bressoud [33] all provide further discussion of algorithms for factoring and computing discrete logarithms, and the latter two books are highly recommended for the reader wishing to understand the state-of-the-art.

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 [100] and Shoup [114].

Exercises

8.1 Here we show how to solve the discrete logarithm problem in a cyclic group of order $q = p^e$ in time $\mathcal{O}(\mathsf{polylog}(q) \cdot \sqrt{p}\,)$. We are given as input a generator g of known order p^e and a value y, and want to compute $x = \log_g y$. Note that p can be computed easily from q (see Exercise 7.9 in Chapter 7).

(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 $e = \mathsf{polylog}(q)$.)

8.2 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 $\mathcal{O}\left(\mathsf{polylog}(q) \cdot \sum_{i=1}^{k} e_i \sqrt{p_i}\right) = \mathcal{O}\left(\mathsf{polylog}(q) \cdot \max\{\sqrt{p_i}\,\}\right)$.

8.3 (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 $\mathbb{Z}_N$.

Chapter 9

Private-Key Management and the Public-Key Revolution

9.1 Limitations of Private-Key Cryptography

9.1.1 The Key-Management Problem

Until now we have learned that given shared, secret keys it is possible to solve the primary problem of cryptography: how to securely communicate over an insecure channel. However, we have not really dealt at all with the question of how shared, secret keys can be obtained in the first place. Clearly, they cannot be sent over an insecure communication channel, because an eavesdropping adversary could then catch them *en route*.

Key distribution and setup. 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 being used.)

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 (perhaps by flipping coins) 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 in 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 the other employees* would have to share a new secret key with her. This would be especially problematic if the company is large and spread over a number of different physical locations.

A partial solution in the above 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 ℓ random keys $k_1, \ldots, k_\ell$, give these keys to the new employee (in person), 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 even give a complete solution, as the controller knows all the keys shared by all employees. A dishonest controller will be able to decrypt all inter-employee communication.

Key storage and secrecy. Consider again the aforementioned work environment where each pair of employees shares a secret key. This means that when there are U employees, the number of secret keys in the system is $\binom{U}{2} = \Theta(U^2)$. Perhaps more importantly, this means that every employee must hold $U-1$ secret keys so that it can communicate with all other employees. In fact, the situation is far worse because employees may need keys to communicate securely with *remote resources* as well (e.g., 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 creates a significant logistical problem. Second, all these secret keys must be stored in a secure place. 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 multiple keys on a personal computer is not a reasonable solution.

Secret keys must be stored in a secure fashion irrespective of the number of keys each party holds. When only a few keys need to be stored, however, there are good solutions for doing this. The typical solution today is to use something called a smartcard, which is essentially a highly protected piece of hardware where secret cryptographic keys are stored. Cryptographic computations using the secret keys take place on the smartcard, ensuring that parties' secret keys never find their way onto their insecure personal computers. Smartcards are typically quite limited in memory, making them impractical for storing hundreds (if not thousands) of keys. Of course, in principle similar solutions can be found. In practice, though, a system with many secret keys is simply more vulnerable to attack (you can hide a needle in a haystack but it's hard to hide thousands of needles in a haystack).

Once again, organizations with many resources like governments and armies are able to solve these problems. 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 special highly-protected and locked room where all of the keys were stored. He then found the appropriate key (by its identity sent with the message) and used the key to decrypt the message.[1] The point of this story is that governments and organizations on that scale could use private-key encryption in order to solve their problems. However, the solutions were very costly, do not scale well, and are not suitable for settings that are typical in industry or for personal use.

The limitations of private-key cryptography. As we have discussed, private-key cryptography can be used in "closed" systems, where it is possible to distribute secret keys via physical means. However, there are two very important points to note. First, even within closed systems, private-key cryptography is difficult to deploy and maintain, and many security risks arise from the necessity to manage so many keys. The second point to realize is that in many settings it is not possible to use private-key cryptography at all. For example, when encryption is needed for making a purchase over the Internet, or sending email to a colleague in another country (who we may never have met), private-key cryptography just does not provide a solution. Thus in "open" or "ad-hoc" systems, different solutions are needed. 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 and/or have transient interactions.*

9.1.2 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 its inapplicability to 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.

Key distribution centers. Consider again the case of a large organization where all pairs of employees must be able to communicate securely. The

[1] For obvious reasons, the description told of this job was very vague. We therefore cannot guarantee perfect accuracy. In any case, the system worked essentially as described here.

solution in which each pair of employees shares a key results in a huge proliferation of keys in the system. A different approach is to rely on the fact that all employees may *trust* the same entity (at least with respect to the security of work-related information), say the IT manager of the organization. It is therefore possible for the IT manager to set up a single server — called a *key distribution center* or KDC for short — that acts as an *intermediary* between employees that wish to communicate, and *distributes* keys to the employees. This server 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'`. The KDC then chooses a new random secret key (called a *session key*) and sends this key to Alice encrypted under Alice's key and to Bob encrypted under Bob's key. Once Alice and Bob receive 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 following 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 a party 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). In short, this approach makes private-key cryptography practical in a single organization, where there is one person 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 above system has a single point of failure: if the KDC crashes, all 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.

   A simple solution is to replicate the KDC. This works (and is essential in practice), however it is important to be aware that the existence of more KDCs means that there are now more points of attack on the system. Furthermore, it becomes more difficult to add a new employee since updates to any KDC must be securely propagated to all others.

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 via 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. The protocol was designed in this way due to the fact that Bob may not be online and this will cause a problem for the KDC who will have to decide when to timeout. By sending both encrypted keys to Alice, the KDC is relieved of unnecessary work. 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.

In practice, protocols like the Needham-Schroeder protocol are widely used. In many cases Alice and Bob are not two individuals, but Alice may be an individual and Bob a resource. For example, Alice may wish to read from a protected disk on some server. Then, 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 the secret key that Alice shared with the KDC is usually a short, memorizable password (because most users don't have smartcards to store 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.

9.2 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 basic problem of key distribution in open systems where there is no secure channel. (Note that in the KDC setting, we assume a "secure channel" that is used to share a key between an employee and the KDC when the employee first joins the organization.) In particular, KDCs do not help in Internet-like settings. In order to deal with this, 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" [50]. 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 due to their work. Until 1976, it was well accepted that encryption could not be carried out without first sharing a secret key, as we have seen so far in this book. However, Diffie and Hellman observed that there is asymmetry in the world: there are certain operation that can be

easily carried out but cannot be easily inverted. For example, many 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, we have seen that it is easy to multiple 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 opens the possibility for constructing an encryption scheme for which the encryption and decryption keys are different. Furthermore, and most importantly, the security of the encryption scheme is preserved even against an adversary who knows the encryption key. Given such a scheme, it is possible to publicize the encryption key (say, in the newspaper), enabling everyone to send encrypted messages. Of course, the main point is that the encryption scheme remains secure even though the encryption key has been publicized. Such encryption schemes are called *asymmetric* or *public-key* (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* and the decryption key is called the *private key*.

It is clear why this helps with key distribution. With a public-key encryption scheme it is possible to post one's public keys on one's webpage, or send the public key to another party via email, without ever having to arrange a physical meeting.[2] Furthermore, it is no longer necessary to store many secret keys. Rather, each party needs to store only his or her own private key; other parties' public keys can either be obtained when needed, or stored in a *non-secure* (i.e., publicly-readable) fashion. (Clearly, however, the integrity of stored public keys must be preserved.) Given the above, and the fact that private-key encryption is not implementable in many settings (like today's Internet), the invention of public-key encryption was indeed a revolution in cryptography. Indeed, 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; only with the advent of public-key cryptography did the use of cryptography spread to the masses.[3]

Diffie and Hellman actually introduced three distinct public-key (or asymmetric) primitives. The first is that of public-key encryption, described above; we will study this notion in Chapter 10. The second is a public-key analogue of message authentication codes, called *digital signatures schemes* and discussed in Chapter 12. As with MACs, a digital signature scheme is used to prevent any undetected tampering of the signed message. In contrast to MACs, however, authenticity of a message can be verified by anyone who knows the

[2] We stress that this is actually not as simple as it seems. Nevertheless, the point is that public keys can be distributed over a *public* (but authenticated) channel.

[3] Of course, until the early '80s computers were also not in widespread use in the public domain. Nevertheless, cryptography could not have moved to the public domain until the problem of key distribution was solved.

public key of the sender, whereas only the owner of the corresponding private key can generate a valid digital signature. This turns out to have far-reaching ramifications. Specifically, it is possible to take a document that was digitally signed by a party Alice and present it to a third party, such as 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. Another highly important application of digital signatures is 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 have any secret information can generate a secret key by communicating over an open network. The main property here is that an eavesdropping adversary who sees all the messages sent over the communication line does not learn anything about the secret key. (Such a protocol cannot be constructed using only private-key techniques.) Stopping to think about it, this 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) won't learn anything about!

The main difference between key exchange and encryption is that the former requires both parties to be online. This is in contrast to encryption that is a non-interactive process and is thus more appropriate for some applications. Secure email, for example, requires encryption because the recipient is not necessarily online when the email message is sent.

Although Diffie and Hellman introduced all three of the above primitives, they only presented a construction for the problem of key exchange. 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 [110]. Variants of the schemes they presented are the most widely used today. Interestingly, in 1985 an encryption scheme based on the Diffie-Hellman key-exchange protocol was presented [60]. Thus, although Diffie and Hellman did not succeed in constructing a (non-interactive) public-key encryption scheme, they came very close. We describe the Diffie-Hellman key-exchange protocol in the next section.

History. It is fascinating to read about the history leading to the public-key revolution initiated by Diffie and Hellman. First, it is important to note that similar ideas were floating around at the time. Another prominent researcher who was doing similar and independent work at the same time was Ralphe Merkle (Merkle is considered by many to be a co-inventor of public-key cryptography, although he published after Diffie and Hellman). Another in-

teresting fact is that it seems that a public-key encryption scheme was known to the intelligence world (at the British intelligence agency GCHQ) in the early 1970s. It appears, however, that although the underlying mathematics of public-key encryption may have been discovered earlier, the widespread ramifications of this new technology were not appreciated until Diffie and Hellman came along.

At the time that their work was carried out, Diffie and Hellman (and others working on and 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 it was considered highly problematic. Indeed, Hellman tells a story where he personally gave a conference presentation of joint work with Ralph Merkle and Steve Pohlig, even though they were students and he was a professor (the usual practice is that students who co-author a paper are the ones to present it). The reason for this reversal of protocol was that, under the advice of Stanford's general counsel, it was 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. We remark that limitations on the export of cryptographic *implementations* from the US continue until today. However, since 2000 they have been greatly reduced and are hardly felt at all.

9.3 Diffie-Hellman Key Exchange

In this section we will present the Diffie-Hellman key exchange protocol and will prove its security in the presence of eavesdropping adversaries. We will *not* present definitions of security of key exchange for active adversaries who may intercept and modify messages sent between the parties, as this material is beyond the scope of this book.

The setting and definition of security: eavesdropping adversaries. We will consider a setting with two parties Alice and Bob that run some protocol in order to exchange a key; we denote the protocol by Π (thus Π is the set of instructions for Alice and Bob in the protocol). The input of both Alice and Bob in the protocol is the security parameter in unary form, and they use random coins in their computation. Denote by r_A the random coins used by Alice and by r_B the random coins used by Bob. Furthermore, let $\mathsf{output}_{A,\Pi}(1^n, r_A, r_B)$ and $\mathsf{output}_{B,\Pi}(1^n, r_A, r_B)$ denote the respective outputs of Alice and Bob from Π, upon input 1^n and respective random coins r_A and r_B. Finally, let $\mathsf{transcript}_{\Pi}(1^n, r_A, r_B)$ denote the transcript of all messages sent by Alice and Bob in an execution of Π, upon input 1^n and respective

random coins r_A and r_B. We assume that this output takes the form of an n-bit key that is supposed to be shared by Alice and Bob. We are now ready to present the definitions.

The first definition is a correctness requirement and it states that, except with negligible probability, Alice and Bob must agree on the same key.

DEFINITION 9.1 *A protocol Π for key exchange is called* correct *if there exists a negligible function* negl *such that for every n,*

$$\Pr\left[\mathsf{output}_{A,\Pi}(1^n, r_A, r_B)\right] \neq \Pr\left[\mathsf{output}_{B,\Pi}(1^n, r_A, r_B)\right] \leq \mathsf{negl}(n)$$

where the probabilities are taken over the choice of r_A and r_B.

We now move on to define security. Intuitively, a key exchange protocol is secure if the key output by Alice and Bob is completely secret to an eavesdropping adversary. This is formulated by giving an adversary the transcript of a protocol execution and seeing if it can distinguish between the key output by the parties and a completely random key. In order to define this, we present an experiment that is somewhat reminiscent of the experiment used to define security of encryption. Let Eve be an adversary and n the security parameter. We have the following experiment:

The eavesdropping key exchange experiment $\mathsf{KE}^{\mathsf{eav}}_{\mathsf{Eve},\Pi}(n)$:

1. *Random strings r_A and r_B are chosen of the appropriate length, as defined in protocol Π.*
2. *A random bit $b \leftarrow \{0,1\}$ is chosen; if $b = 0$ then set $k := \mathsf{output}_{A,\Pi}(1^n, r_A, r_B)$, and if $b = 1$ then $k \leftarrow \{0,1\}^n$.*
3. *The adversary* Eve *is given for input the security parameter 1^n, the transcript $\mathsf{transcript}_{\Pi}(1^n, r_A, r_B)$ and the key k, and outputs a bit b'.*

   Formally, $b' \leftarrow \mathsf{Eve}\left(1^n, \mathsf{transcript}_{\Pi}(1^n, r_A, r_B), k\right)$.
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. (In case $\mathsf{KE}^{\mathsf{eav}}_{\mathsf{Eve},\Pi}(n) = 1$, we say that* Eve succeeds.*)*

Observe that k could be set to either $\mathsf{output}_{A,\Pi}(1^n, r_A, r_B)$ or $\mathsf{output}_{B,\Pi}(1^n, r_A, r_B)$ (when $b = 0$) and it wouldn't make a difference because we are only interested in correct protocols.

The transcript and output value are generated using the same r_A and r_B and are therefore correlated as in a real execution. The adversary Eve succeeds in the experiment if it can guess whether the key k it was given is the "correct" one, or it is completely random and independent of the transcript. As expected, we say that the protocol Π is secure if the adversary succeeds with probability that is at most negligibly greater than $1/2$. That is:

DEFINITION 9.2 *A key exchange protocol* Π *is* secure in the presence of eavesdropping adversaries *if for every probabilistic polynomial-time adversary* Eve *there exists a negligible function* negl *such that*

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

We remark that the aim of a key exchange protocol is to use the output key k for encrypting and possibly MACing messages that are now to be exchanged between the parties. The fact that the above definition guarantees that one can use the output key for encryption and MAC, and security is preserved, needs a formal proof. However, such a proof is beyond the scope of this book. Rather, we rely on intuition here: if the output key looks exactly like a random one, then clearly it can be used for encryption, MAC or any other cryptographic task for which a random key is needed. (We warn against relying on intuition alone in cryptography. However, we do not rely on intuition and once again note that there is a proof that this suffices.)

The Diffie-Hellman key exchange protocol. We will now describe the Diffie-Hellman key exchange protocol that appeared in their original paper. The protocol is based on the difficulty of the discrete log problem (see Chapter 7), and can be formally proven secure under the decisional Diffie-Hellman (DDH) problem.

Before motivating the security of the protocol, we show that it is correct. In order to see this, notice that $h_2 = g^y$ and thus $h_2^x = g^{yx} = g^{xy}$. Likewise, $h_1 = g^x$ and so $h_1^y = g^{xy}$. Therefore $h_2^x = h_1^y$ meaning that Alice and Bob output the same key k. Next, regarding security, notice that an eavesdropping adversary Eve cannot carry out the same operation as Alice or Bob in order to compute the key k. This is because Eve does not know x or y and cannot compute them because this involves solving the discrete logarithm problem (which is assumed to be hard). However, who says that the only way of computing k is to first find x or y? It may be possible to take $h_1 = g^x$ and $h_2 = g^y$ and compute $k = g^{xy}$ without ever obtaining x or y. This task is exactly the computational Diffie-Hellman problem and it too is assumed to be hard. However, with a bit more thought, this is also not enough to guarantee security. Specifically, it may be hard to compute g^{xy} but it may be easy to distinguish it from a random group element. This is not a merely theoretical concern, because it may be possible to obtain half of the bits of g^{xy} and no more. In such a case, it is indeed hard to solve the computational Diffie-Hellman problem and thus hard to obtain the entire key k. However, possession of half the key may suffice for breaking the encryption scheme (or for whatever task the key is used). We therefore need a stronger assumption, and that is that the output key g^{xy} is not just hard to compute but is also hard to distinguish from a random element. This is exactly the decisional Diffie-

CONSTRUCTION 9.3 Diffie-Hellman key exchange.

- **Common input:** The security parameter n
- **The protocol:**
  1. Alice generates the description of a group $\mathbb{G}$ and generator g using a group-generation algorithm $\mathcal{G}$ with input 1^n, and sends the result to Bob. Let m denote the order of g.[a]
  2. Alice chooses a random index $x \in \{1, \ldots, m-1\}$ and computes $h_1 = g^x$. Alice sends h_1 to Bob.
  3. Bob chooses a random index $y \in \{1, \ldots, m-1\}$ and computes $h_2 = g^y$. Bob sends h_2 to Alice. (Note that this can be computed and sent before Bob receives h_1 from Alice.)
  4. Alice outputs $k = h_2^x$.
  5. Bob outputs $k = h_1^y$.

[a] It is possible to have the protocol assume that both parties have the description of $\mathbb{G}$ and g ahead of time. In such a case, Alice and Bob's messages can be generated independently of each other.

Hellman assumption and is thus what is required for proving the security of the protocol.

Before proceeding to the proof, we remark that the output of Alice and Bob from the protocol is an element in $\mathbb{G}$. The representation of such an element may be easily distinguished from a random string of the same length. For example, in the group $\mathbb{Z}_p^*$ for a prime p, the most significant bit may be 0 with very high probability. We thus modify the experiment so that if $b = 1$ the adversary is given a random group element $h \leftarrow \mathbb{G}$ and not a uniformly distributed string of the same length. We denote this modified experiment by $\mathsf{KE}^{\mathsf{eav}\prime}$. We now proceed to the proof:

THEOREM 9.4 *Assuming that the decisional Diffie-Hellman problem is hard relative to the group generation algorithm $\mathcal{G}$, the Diffie-Hellman key exchange protocol of Construction 9.3 is correct and secure in the presence of eavesdropping adversaries.*

PROOF Correctness has already been proved in the discussion above. We prove security. By the definition of the experiment $\mathsf{KE}^{\mathsf{eav}\prime}$, the adversary Eve receives $(\mathbb{G}, g, h_1, h_2, k)$, where $(\mathbb{G}, g)$ are generated according to $\mathcal{G}$, and $k = h_1^y = h_2^x$ if $b = 0$ and $k \leftarrow \mathbb{G}$ is a random element if $b = 1$. Writing this out in full we have that this input of Eve equals $(\mathbb{G}, g, g^x, g^y, g^{xy})$ if $b = 0$ and $(\mathbb{G}, g, g^x, g^y, h)$ if $b = 1$ (where h is a random element in $\mathbb{G}$ and x and y

are chosen randomly as in the protocol). Thus, distinguishing between these two elements is equivalent to solving the decisional Diffie-Hellman problem. Formally, let ε be a function such that $\Pr\left[\mathsf{KE}^{\mathsf{eav}\prime}_{\mathsf{Eve},\Pi}(n)=1\right] = \frac{1}{2}+\varepsilon(n)$. Noting that $\Pr[b=0]=\Pr[b=1]=1/2$, we have

$$\begin{aligned}
&\Pr\left[\mathsf{KE}^{\mathsf{eav}\prime}_{\mathsf{Eve},\Pi}(n)=1\right] \\
&= \frac{1}{2}\cdot\Pr\left[\mathsf{KE}^{\mathsf{eav}\prime}_{\mathsf{Eve},\Pi}(n)=1 \mid b=0\right] + \frac{1}{2}\cdot\Pr\left[\mathsf{KE}^{\mathsf{eav}\prime}_{\mathsf{Eve},\Pi}(n)=1 \mid b=1\right] \\
&= \frac{1}{2}\cdot\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,g^{xy})=0] + \frac{1}{2}\cdot\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,h)=1]
\end{aligned}$$

and therefore

$$\frac{1}{2}\cdot\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,g^{xy})=0] + \frac{1}{2}\cdot\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,h)=1] = \frac{1}{2}+\varepsilon(n)$$

or equivalently

$$\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,g^{xy})=0] + \Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,h)=1] = 1+2\varepsilon(n).$$

Noting that $\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,g^{xy})=0] = 1-\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,g^{xy})=1]$ we have that

$$1-\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,g^{xy})=1] + \Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,h)=1] = 1+2\varepsilon(n)$$

and so

$$\Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,h)=1] - \Pr[\mathsf{Eve}(\mathbb{G},g,g^x,g^y,g^{xy})=1] = 2\varepsilon(n).$$

By the assumed hardness of the decisional Diffie-Hellman assumption relative to the group generation algorithm $\mathcal{G}$, the above difference must be negligible and so ε must also be a negligible function. We therefore conclude that $\Pr\left[\mathsf{KE}^{\mathsf{eav}\prime}_{\mathsf{Eve},\Pi}(n)=1\right]$ is at most negligibly greater than $1/2$ and so the protocol is secure in the presence of eavesdropping adversaries. ■

Random elements versus random strings. We have proven that the output of Alice and Bob from the Diffie-Hellman key exchange protocol looks the same as a random group element (to probabilistic polynomial-time adversaries). However, in most cases, the secret key that Alice and Bob wish to use must be a uniformly distributed (or pseudorandom) string of some length (as required by the original experiment $\mathsf{KE}^{\mathsf{eav}}$). Thus, the random group element must be converted into a string that can be used for encryption. Such an operation can be carried out and the function used for the conversion is called a "randomness extractor". We will not describe how this is done here.

Insecurity of Diffie-Hellman key exchange for active adversaries. So far we have considered the case of an eavesdropping adversaries. Now,

although eavesdropping attacks are by far the most common (as they are so easy to carry out), they are in now way the only type of attack. Rather, it is possible to carry out active attacks in which the adversary sits in between Alice and Bob, intercepts their messages and modifies them or sends messages of its own. Such an adversarial attack is called a *man-in-the-middle attack* and any key exchange protocol to be used today must be secure also in the presence of adversaries carrying out such an attack. We will not define security for this case as it is rather involved and we will also not show how to achieve key exchange in the presence of man-in-the-middle attacks. However, we do remark that the Diffie-Hellman protocol is *not* secure against man-in-the-middle attacks. In fact, a man-in-the-middle adversary can act so that Alice and Bob conclude with different keys k_1 and k_2 that the adversary knows, and neither Alice nor Bob know that any attack was carried out. We leave the details of this attack as a highly recommended exercise. We remark that 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 was the first demonstration that asymmetric techniques (and number-theoretic problems) can be used to alleviate the problems of key distribution in cryptography. We also remark that using only private-key (symmetric) cryptographic techniques, it is impossible to achieve secure key exchange even in the presence of eavesdropping adversaries. Finally, we note that there are solutions to the problem of key exchange and key distribution in the presence of active adversaries who can carry out man-in-the-middle attacks. These solutions are discussed below in Section 12.7.

Diffie-Hellman in practice. The Diffie-Hellman key exchange protocol in its basic form, as described above, is typically not used in practice. This is due to its insecurity in the face of man-in-the-middle attacks. However, it does form the basis of other key exchange protocols that are resilient to man-in-the-middle attacks; these more advanced protocols take the core of the Diffie-Hellman protocol and add to it in order to achieve the desired resilience. One notable example of a standardized protocol that uses Diffie-Hellman is IPsec. Loosely speaking, the IPsec standard is typically used to connect servers so that they can communicate securely over an insecure channel. The key exchange protocol that stands at the heart of IPsec is a version of *authenticated* Diffie-Hellman, that as we have mentioned, takes the basic Diffie-Hellman protocol and add steps so as to thwart man-in-the-middle attacks. We therefore conclude that although Diffie and Hellman did not succeed in constructing an encryption or digital signature scheme, their key-exchange protocol is in widespread use. Thus, in addition to their conceptual contribution to cryptography and its enormous impact, they also constructed a protocol that is in use until today (and will most likely be in use for many many years to come).

References and Additional Reading

We have only very briefly discussed the problems of key distribution and key management in general. For more information, we recommend looking at some of the Network Security literature. Our favorite book is [84] and it provides an excellent treatment of the different KDC protocols, what they aim to achieve and how they work.

We highly recommend reading the original paper by Diffie and Hellman [50]. The history of the development of public-key cryptography is a fascinating one and much information can be found on the Internet. An interesting book that focuses on the political and historical aspects of the public-key revolution is that of [89].

Exercises

9.1 Describe in detail a man-in-the-middle attack on the Diffie-Hellman key exchange protocol. Show how an adversary can act so that at the end of the protocol Alice has a key k_1 (known to the adversary) and Bob has a key k_2 (known to the adversary). In addition to the above, show how an adversary can act so that if Alice and Bob continue by carrying out an encrypted conversation (using their keys obtained from the protocol), then the adversary obtains all the plaintext messages sent and Alice and Bob don't know that any attack was carried out.

9.2 Consider the following key-exchange protocol that is based on the one-time pad and perfect secrecy:

(a) Alice chooses a random key k and a random string r both of length n, and sends $s = k \oplus r$ to Bob.

(b) Bob chooses a random string t of length n 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 break).

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 two 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 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. In the setting of public-key encryption, one party (the *receiver*) generates a *pair* of keys (pk, sk), called the *public key* and the *private key*, respectively. The public key can be used by a *sender* to encrypt a message for the receiver; the receiver can then use 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 from Alice to Bob may be public, but is assumed to be authenticated. (See Section 12.7 for a discuss of how public key can be distributed over unauthenticated channels.) An example is the "shouting-across-a-room" scenario mentioned earlier, if we imagine that Alice shouts her public key (bit-by-bit, say) and then Bob shouts his ciphertext.

An alternate 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 privacy of public-key encryption cannot rely on secrecy of pk, but must instead rely on 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). Though 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 the key; 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 systems 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 these 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 discussion in the previous chapter):

- The most important advantage is that public-key encryption solves (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 when 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 roughly 2–3 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 (as might be the case for a server processing credit-card transactions for an on-line merchant) may be prohibitive. 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 and Active Adversaries

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 all honest parties, 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 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

[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.

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 12.7 how reliance on a trusted third party can be used to address such attacls.

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 give.) 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 12.7), and it is therefore convenient (and useful) to de-couple 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.

DEFINITION 10.1 *A* public-key encryption scheme *is a tuple of probabilistic, polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *that satisfies the following:*

1. *Algorithm* Gen *takes as input a 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. *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* $c \leftarrow \mathsf{Enc}_{pk}(m)$*.*

3. *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* $m := \mathsf{Dec}_{sk}(c)$*.*

[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.

We require that for every n, every (pk, sk) output by $\mathsf{Gen}(1^n)$, and every message m in the appropriate underlying plaintext space, it holds that

$$\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m)) = m.$$

Recall from the previous chapter how a public-key encryption scheme is used. One party, the *receiver*, runs $\mathsf{Gen}(1^n)$ to obtain keys (pk, sk). The public key pk is then made available for anyone who wants to encrypt a message for this receiver; for example, the receiver may post their key on their webpage or store it in some public repository of public keys. We assume that anyone wishing to encrypt a message for this receiver is able to obtain a legitimate copy of this receiver's public key. (As we have already noted, this is a non-trivial problem and one that we will return to in Section 12.7. Using this assumption allows us to de-couple the problem of obtaining the correct key from the problem of using the key once it is obtained.) A sender, holding the public key pk of the receiver, can encrypt their message m by computing $c \leftarrow \mathsf{Enc}_{pk}(m)$. The receiver, upon receipt of c, can recover the message by computing $m := \mathsf{Dec}_{sk}(c)$.

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. Recall from the previous chapter that 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.

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 that the receiver can recover the bit-string being encrypted). As we will see later (cf. Proposition 10.11), a public-key encryption scheme for the plaintext space $\{0,1\}$ extends immediately to give an encryption scheme for $\{0,1\}^*$.

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 3.9 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 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 *with* $|m_0| = |m_1|$. *(These messages must be in the plaintext space associated with* pk*.)*
3. *A random bit* $b \leftarrow \{0,1\}$ *is chosen, and then the 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*

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

where the probability is taken over the random coins used by $\mathcal{A}$, *as well as the random coins used to generate* (pk, sk), *choose* b, *and encrypt* m_b.

The main difference between the above definition and Definition 3.9 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 in the previous chapter, 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. (Indeed, the recipient makes no effort to keep his public key hidden; to the contrary, he may have widely publicized his public key.)

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 3.22. 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 *with* $|m_0| = |m_1|$. *(These messages must be in the plaintext space associated with* pk*.)*
3. *A random bit* $b \in \{0,1\}$ *is chosen, and then the ciphertext* $c \leftarrow \mathsf{Enc}_{pk}(m_b)$ *is computed and given to* $\mathcal{A}$. *We call* c *the* challenge ciphertext. $\mathcal{A}$ *continues to have access to* $\mathsf{Enc}_{pk}(\cdot)$.
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 10.4 *Public-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *has* indistinguishable encryptions under chosen-plaintext attacks *(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:

PROPOSITION 10.5 *If public-key encryption scheme* Π *has indistinguishable encryptions in the presence of an eavesdropper, then* Π *also has indistinguishable encryptions under chosen-plaintext attacks.*

This is in contrast to the private-key setting, where the above does not hold. (See Claims 3.20 and 3.23.) Further differences from the private-key setting that follow almost immediately as a consequence of the above are discussed next.

Impossibility of Perfectly-Secure 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., in-

cluding the pubic 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 does not exist. (We stress that this holds 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. Because of the equivalence, in the public-key setting, between CPA-security and indistinguishability of encryption in the presence of an eavesdropper, 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. First, it is important to understand that 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 scheme leaves one 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.

Second, though the result seems deceptively simple, for a long time *many real-world systems were designed using deterministic public-key encryption.* In fact, 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 looking at things the right way for the first time (even if seemingly simple in retrospect), and pinning down one's intuition in a formal definition, should not be underestimated.

10.2.2 Security for Multiple Encryptions

As in Chapter 3, it is important also 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 such a definition is automatically equivalent to a definition in which chosen-plaintext attacks are also allowed. We then prove that the two definitions are *equivalent*. We can therefore prove schemes secure with respect to eavesdropping on the encryption of a single message (which is a simpler definition that is easier to work with), and obtain security with respect to eavesdropping on the encryptions of multiple messages (which is a stronger definition that more accurately models adversarial attacks).

We remark that an analogous result in the private-key setting was stated, but not proved, as Claim 3.23. That claim refers to security under chosen-plaintext attacks, but as we have seen already chosen-plaintext attacks are "for free" in the public-key setting.

Consider the following experiment defined for public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and adversary $\mathcal{A}$:

Eavesdropping on multiple encryptions $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$:

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, and* $\mathcal{A}$ *is given the vector* $\vec{C} = (\mathsf{Enc}_{pk}(m_b^1), \ldots, \mathsf{Enc}_{pk}(m_b^t))$*. The 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 *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. Specifically, 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 $(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2))$ while if $b = 1$ the adversary is given $(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2))$. We show that there exists a negligible function negl such that

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] \\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\ &\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1],\end{aligned}$$

where the equality follows by conditioning on the two possible values of b. Since this holds for every probabilistic polynomial-time adversary $\mathcal{A}$, this completes the proof.

Consider what would happen if $\mathcal{A}$ were given $(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2))$. 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*

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}(n) &\geq \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\ &\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 1].\end{aligned}$$

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 $b = 0$ in that experiment then $\mathcal{A}'$ is given $\mathsf{Enc}_{pk}(m_0^2)$. Furthermore,

$$\Pr[\mathcal{A}'(\mathsf{Enc}_{pk}(m_0^2)) = 0] = \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0].$$

On the other hand, when $b = 1$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ then $\mathcal{A}'$ 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 security of Π (in the sense of single-message indistinguishability), it must be the case that there exists a negligible function negl such that

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}_1(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1] \\ &= \frac{1}{2}\Pr[\mathcal{A}'(\mathsf{Enc}_{pk}(m_0^2)) = 0] + \frac{1}{2}\Pr[\mathcal{A}'(\mathsf{Enc}_{pk}(m_1^2)) = 1] \\ &= \frac{1}{2}\cdot\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\ &\quad + \frac{1}{2}\cdot\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1],\end{aligned}$$

completing the proof of the claim. ■

By an exactly analogous argument, we 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

$$\begin{aligned}1 + \mathsf{negl}(n) &\geq \frac{1}{2}\cdot\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] \\ &\quad + \frac{1}{2}\cdot\Big(\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 1] \\ &\qquad + \Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_1^2)) = 0]\Big) \\ &\quad + \frac{1}{2}\cdot\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1] \\ &= \frac{1}{2}\cdot\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_0^1), \mathsf{Enc}_{pk}(m_0^2)) = 0] + \frac{1}{2} \\ &\quad + \frac{1}{2}\cdot\Pr[\mathcal{A}(\mathsf{Enc}_{pk}(m_1^1), \mathsf{Enc}_{pk}(m_1^2)) = 1],\end{aligned}$$

implying that

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

as desired.

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 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 independently random coins, and so the above actually represents a *distribution* over vectors containing t ciphertexts. Using this notation, we have

$$\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] = \frac{1}{2} \cdot \Pr[\mathcal{A}(\vec{C}^{(t)}) = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}(\vec{C}^{(0)}) = 1].$$

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 can view $\mathcal{A}'$ as "guessing" an index $i \in \{0, \ldots, t\}$ for which $\mathcal{A}$ can distinguish between $\vec{C}^{(i)}$ and $\vec{C}^{(i-1)}$.

In experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$, when $b = 0$ and $i = i^*$ then $\mathcal{A}'$ is given $\mathsf{Enc}_{pk}(m_0^{i^*})$ and $\mathcal{A}$ is run on input distributed according to $\vec{C}^{(i^*)}$. So,

$$\begin{aligned}\Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] &= \sum_{i^*=1}^{t} \Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0 \bigwedge i = i^*] \cdot \Pr[i = i^*] \\ &= \sum_{i^*=1}^{t} \frac{1}{t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 0].\end{aligned}$$

On the other hand, when $b = 1$ and $i = i^*$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$, then $\mathcal{A}'$ is given $\mathsf{Enc}_{pk}(m_1^{i^*})$ and $\mathcal{A}$ is run on input distributed according to $\vec{C}^{(i^*-1)}$. So,

$$\begin{aligned}\Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1] &= \sum_{i^*=1}^{t} \Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1 \bigwedge i = i^*] \cdot \Pr[i = i^*] \\ &= \sum_{i^*=0}^{t-1} \frac{1}{t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 1].\end{aligned}$$

(Note that the indices of summation have been shifted.)

By assumption, Π has indistinguishable encryptions in the presence of an eavesdropper and so that there exists a negligible function negl such that

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1] \\ &= \frac{1}{2}\Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] + \frac{1}{2}\Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1] \\ &= \sum_{i^*=1}^{t} \frac{1}{2t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 0] + \sum_{i^*=0}^{t-1} \frac{1}{2t} \cdot \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 1] \\ &= \frac{1}{2t} \cdot \sum_{i^*=1}^{t-1} \Big(\Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 0] + \Pr[\mathcal{A}(\vec{C}^{(i^*)}) = 1] \Big) \\ &\quad + \frac{1}{2t} \cdot \Big(\Pr[\mathcal{A}(\vec{C}^{(t)}) = 0] + \Pr[\mathcal{A}(\vec{C}^{(0)}) = 1] \Big) \\ &= \frac{t-1}{2t} + \frac{1}{t} \cdot \Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1].\end{aligned}$$

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 $\mathcal{A}$ was 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:

$$\mathsf{Enc}'_{pk}(m) = \mathsf{Enc}_{pk}(m_1), \ldots, \mathsf{Enc}_{pk}(m_t),$$

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'*.*

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 (that is, it has indistinguishable *multiple* encryptions under *chosen-plaintext attack*). 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" (both in our discussion as well as in our 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

In the discussion prior to Proposition 10.11, we showed how any CPA-secure public-key encryption scheme for 1-bit messages can be used to obtain a CPA-secure encryption scheme for messages that are arbitrarily long. Unfortunately, encrypting a t-bit message using the approach shown there 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. Is it possible to do better?

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 will improve 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:

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 intuitively 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* 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.

Of course, the above two steps can be performed in "one shot" by having the sender transmit the single 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*.

A formal description of the above is given as Construction 10.12. The construction assumes that Π includes $\{0,1\}^n$ in the underlying plaintext space; the plaintext space for Π^{hy} is identical to the plaintext space of Π'.

CONSTRUCTION 10.12

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme, and let $\Pi' = (\mathsf{Enc}', \mathcal{D}')$ be a private-key encryption scheme. Construct public-key encryption scheme $\Pi^{\mathsf{hy}} = (\mathsf{Gen}^{\mathsf{hy}}, \mathsf{Enc}^{\mathsf{hy}}, \mathsf{Dec}^{\mathsf{hy}})$ as follows:

- $\mathsf{Gen}^{\mathsf{hy}}$ is identical to Gen.
- $\mathsf{Enc}^{\mathsf{hy}}_{pk}(m)$ proceeds as follows:
  1. Choose 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 $\langle c_1, c_2 \rangle$.
- $\mathsf{Dec}^{\mathsf{hy}}_{sk}(\langle c_1, c_2 \rangle)$ proceeds 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 previous approach of using Enc to encrypt bit-by-bit? 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. (Refer to the previous chapter for a discussion of the relative efficiency of public- vs. private-key techniques.) 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

$$\frac{\alpha + \beta \cdot t}{t} = \frac{\alpha}{t} + \beta, \tag{10.1}$$

which approaches β as t gets large. In the limit of very long messages, then, the cost per bit of the *public-key* encryption scheme Π^{hy} is the same as the cost per bit of 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 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). Then the total length of a ciphertext in scheme Π^{hy} is

$$\ell + n + |m|. \tag{10.2}$$

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 a feel for the improvement, while actual numbers would depend on a variety of factors.) A typical value for the length n of the key k might be $n \approx 100$. Letting α, as above, denote the cost of public-key encryption of a 100-bit key, a typical public-key encryption scheme might be able to encrypt a message of length t with computational cost $\alpha \cdot \lceil t/1000 \rceil$ and ciphertext length $2 \cdot \lceil t/1000 \rceil$. (That is, it costs the same to encrypt 1 bit or 1000 bits; messages longer than 1000 bits would be parsed into blocks of length at most 1000 bits, and then each block could be encrypted separately.) Finally, if we again let β denote the computational cost (per bit) of private-key encryption then a reasonable approximation is $\beta \approx \alpha/10^6$. The computational cost (per bit) of hybrid encryption for a 1Mb e-mail, as per Equation (10.1), is

$$\frac{\alpha}{10^6} + \frac{\alpha}{10^6} = \frac{\alpha}{5 \cdot 10^5},$$

and the ciphertext length, as per Equation (10.2), would be $10^6 + 2000 + 100 \approx 10^6$ bits. For comparison, block-by-block encryption would have computational cost (per bit)

$$\frac{\alpha \cdot \lceil t/1000 \rceil}{t} \approx \frac{\alpha}{5 \cdot 10^2}$$

and result in a ciphertext of length roughly $2 \cdot 10^6$. Thus, hybrid encryption improves the computational efficiency in this case by a factor of 1000(!), 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. It is interesting to notice here 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 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 suffices for security of the hybrid scheme Π^{hy}.

Before formally proving the security of Π^{hy}, we highlight the overall intuition. Let the notation "$A \stackrel{\text{c}}{\equiv} B$" 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 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

$$\big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)\big) \stackrel{\text{c}}{\equiv} \big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)\big) \tag{10.3}$$

for all m_0, m_1 output by a PPT adversary $\mathcal{A}$, where again pk is output by Gen and in addition the key k is chosen at random. (Equation (10.3) shows that Π^{hy} has indistinguishable encryptions in the presence of an eavesdropper, but by Theorem 10.10 this implies that Π^{hy} is CPA-secure.)

The proof proceeds in three steps; see Figure 10.1. First we prove that

$$\big(pk,\ \mathsf{Enc}_{pk}(k),\ \mathsf{Enc}'_k(m_0)\big) \stackrel{\text{c}}{\equiv} \big(pk,\ \mathsf{Enc}_{pk}(0^n),\ \mathsf{Enc}'_k(m_0)\big) \tag{10.4}$$

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

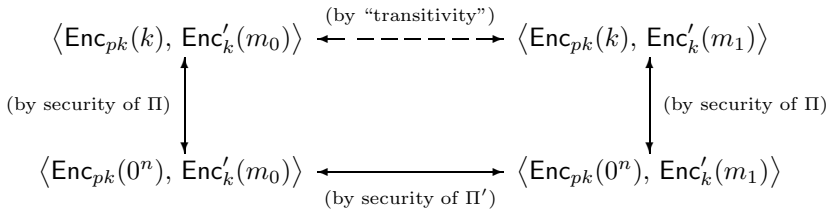

FIGURE 10.1: High-level structure of the proof of Theorem 10.13. The public key pk is left implicit, and is always given to the adversary.

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 also used to 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 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. But if k is encrypted with respect to Π then this is no longer the case.)

Next, we prove that

$$\big(pk, \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)\big) \stackrel{c}{\equiv} \big(pk, \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)\big) \tag{10.5}$$

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

$$\big(pk, \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)\big) \stackrel{c}{\equiv} \big(pk, \mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)\big)\,, \tag{10.6}$$

by relying again on security of Π. Equations (10.4)–(10.6) imply, by transitivity, the desired result in Equation (10.3). (We do not prove that transitivity holds; rather, transitivity in this case is 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 $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi^{\mathsf{hy}}}(n)$. (By Theorem 10.10, once we show that Π^{hy} has indistinguish-

able 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

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}^{\mathsf{hy}},\Pi^{\mathsf{hy}}}(n) = 1] \\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0] \\ &\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1]. \qquad (10.7)\end{aligned}$$

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. $\mathcal{A}_1$ computes $c_2 \leftarrow \mathsf{Enc}'_k(m_0)$. Then $\mathcal{A}_1$ runs $\mathcal{A}^{\mathsf{hy}}(c_1, c_2)$ and outputs the bit b' that is output by $\mathcal{A}^{\mathsf{hy}}$.

In experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_1,\Pi}(n)$, when $b = 0$ then $\mathcal{A}^{\mathsf{hy}}$ 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 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 $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_1,\Pi}(n)$ then $\mathcal{A}^{\mathsf{hy}}$ 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 assumption, Π has indistinguishable encryptions in the presence of an eavesdropper and so there exists a negligible function negl_1 such that

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}_1(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_1,\Pi}(n) = 1] \qquad (10.8) \\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}_1 \text{ outputs } 0 \mid b = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}_1 \text{ outputs } 1 \mid b = 1] \\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0] \\ &\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 1].\end{aligned}$$

Next, consider the following PPT adversary $\mathcal{A}'$ that eavesdrops on a message encrypted using 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 $\mathcal{A}^{\mathsf{hy}}$.

In experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi'}(n)$, when $b = 0$ then $\mathcal{A}^{\mathsf{hy}}$ is given a ciphertext of the form $\langle \mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)\rangle$ for randomly-generated pk and uniform choice of k. We can see that

$$\Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 0].$$

On the other hand, when $b = 1$ then $\mathcal{A}^{\mathsf{hy}}$ 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 assumption, Π' has indistinguishable encryptions in the presence of an eavesdropper and so there exists a negligible function negl' such that

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}'(n) &\geq \Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi'}(n) = 1] \qquad (10.9)\\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}' \text{ outputs } 0 \mid b = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A}' \text{ outputs } 1 \mid b = 1]\\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_0)) = 0]\\ &\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 1].\end{aligned}$$

We will next show something that is exactly analogous to Equation (10.8) (and so the reader is welcome to skip directly to Equation (10.10)). 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 $b = 0$ then $\mathcal{A}^{\mathsf{hy}}$ 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 randomly-generated pk and uniform choice of k. So,

$$\Pr[\mathcal{A}_2 \text{ outputs } 0 \mid b = 0] = \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 0].$$

On the other hand, when $b = 1$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_2,\Pi}(n)$ then $\mathcal{A}^{\mathsf{hy}}$ 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

$$\begin{aligned}\frac{1}{2} + \mathsf{negl}_2(n) &\geq \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_2,\Pi}(n) = 1] \qquad (10.10)\\ &= \frac{1}{2}\cdot\Pr[\mathcal{A}_2 \text{ outputs } 0 \mid b = 0] + \frac{1}{2}\cdot\Pr[\mathcal{A}_2 \text{ outputs } 1 \mid b = 1]\\ &= \frac{1}{2}\cdot\Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(0^n), \mathsf{Enc}'_k(m_1)) = 0]\\ &\quad + \frac{1}{2}\cdot\Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1].\end{aligned}$$

At long last, we come to the conclusion of the proof. Summing Equations (10.8)–(10.10) and using the fact that the sum of three negligible functions is negligible, we see that 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

$$\begin{aligned}&\frac{1}{2} + \mathsf{negl}(n) \geq\\ &\frac{1}{2}\cdot\Big(\Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_0)) = 0] + \Pr[\mathcal{A}^{\mathsf{hy}}(\mathsf{Enc}_{pk}(k), \mathsf{Enc}'_k(m_1)) = 1]\Big),\end{aligned}$$

exactly what we wanted to prove (cf. Equation (10.7)). ■

The above theorem justifies a focus on public-key encryption schemes that can encrypt messages of length n, the security parameter. Of course, it suffices even to 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); the point is that there is not much reason to bother optimizing a public-key encryption scheme once it 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 $ed = 1 \bmod \phi(N)$. Recall from Section 7.2.4 that such an algorithm can be easily constructed from any algorithm $\mathsf{GenModulus}$ that outputs a composite moduli N along with its factorization:

GenRSA

Input: Security parameter 1^n
Output: N, e, d as described in the text

$(N, p, q) \leftarrow \mathsf{GenModulus}(1^n)$
$\phi(N) := (p-1)(q-1)$
find 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 10.14. 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.

CONSTRUCTION 10.14

Let GenRSA be as in the text. Define a public-key encryption scheme as follows:

- $\mathsf{Gen}(1^n)$ runs $\mathsf{GenRSA}(1^n)$ to obtain N, e, and d. The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.
- $\mathsf{Enc}_{pk}(m)$, on input a public key $pk = \langle N, e\rangle$ and a message $m \in \mathbb{Z}_N^*$, computes the ciphertext
$$c := [m^e \bmod N].$$
- $\mathsf{Dec}_{sk}(c)$, on input a private key $sk = \langle N, d\rangle$ and a ciphertext $c \in \mathbb{Z}_N^*$, computes 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. It should be emphasized that this result says nothing about whether the RSA encryption function can be inverted using other means.

In fact, although the textbook RSA scheme 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. 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. We do not prove this result since it follows immediately from the RSA assumption (Definition 7.46). Note that 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).

RSA in Practice

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 method for indicating the length of the message and padded the message to the left with 0s.

A theoretical concern is that (the encoded message) m may not lie in $\mathbb{Z}_N^*$ (i.e., it may be the case that $\gcd(m, N) \neq 1$). Note that even if this occurs, decryption still succeeds as shown in Exercise 7.8 of Chapter 7. In any case if m is chosen at random, then 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 this means that computing eth powers modulo N (as done when encrypting in the "textbook RSA" scheme) requires only two multiplications and hence can be performed very quickly. If e is to be set equal to 3 then p and q must be chosen to satisfy $p, q \neq 1 \bmod 3$.

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 10.14 than as an indication that setting $e = 3$ is a bad choice.

Note that choosing d to be small (that is, changing GenRSA so that d is chosen first and then computing e) in order to speed up decryption 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 to speed up computation of dth roots modulo N (as done when decrypting in the textbook RSA scheme). Specifically, since $[c^d \bmod N] \leftrightarrow (c^d \bmod p, c^d \bmod q)$ the receiver can compute the partial results

$$m_p = c^d \bmod p = c^{[d \bmod (p-1)]} \bmod p$$

and

$$m_q = c^d \bmod q = c^{[d \bmod (q-1)]} \bmod q$$

and then combine these to obtain $m \leftrightarrow (m_p, m_q)$ as discussed in Section 7.1.5.

Example 10.15
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 respect to the 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, simply compute $57 := [250^{147} \bmod 253]$. Alternately, using the Chinese remainder theorem the receiver could compute

$$250^{[147 \bmod 10]} \bmod 11 = 8^7 \bmod 11 = 2$$

and

$$250^{[147 \bmod 22]} \bmod 23 = 20^{15} \bmod 23 = 11.$$

And indeed, $57 \leftrightarrow (2, 11)$. (The desired answer could be recovered from the representation $(2, 11)$ as described in Section 7.1.5.) ♢

10.4.2 Attacks on RSA

To give more of a feel for the RSA problem, as well as to illustrate additional problems with textbook RSA encryption, we describe here various attacks, some of which apply only to Construction 10.14 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 done.

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).

A general attack when small e is used. 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

$$c_1 = m^3 \bmod N_1 \quad \text{and} \quad c_2 = m^e \bmod N_2 \quad \text{and} \quad c_3 = m^3 \bmod N_3.$$

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 can then 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 $\hat{c} < N^*$ with

$$\begin{aligned} \hat{c} &= c_1 \bmod N_1 \\ \hat{c} &= c_2 \bmod N_2 \\ \hat{c} &= c_3 \bmod N_3. \end{aligned}$$

Moreover, using techniques similar to those shown in Section 7.1.5 it is possible to compute $\hat{c}$ efficiently. 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 $1 \leq m < \mathcal{L}$, the time to carry out this attack is linear in $\mathcal{L}$, and so 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 (Section 10.3), where the "message" that is encrypted directly by the public-key component is a random secret key of length n, and so $\mathcal{L} = 2^n$.

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.

We show the attack and then discuss why it works. In the description of the algorithm, we assume that $m < 2^\ell$ and the attacker knows this. The value α is a constant with $\frac{1}{2} < \alpha < 1$.

If an r that is not invertible modulo N is ever encountered, then the factorization of N (and hence m) can be easily computed; we do not bother explicitly writing this in the algorithm above. The time complexity of the

An attack on textbook RSA encryption

Input: Public key $\langle N, e\rangle$; ciphertext c; parameter ℓ
Output: m 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 $x_r = [s^e \bmod N]$ for some r
    return $[r \cdot s \bmod N]$

algorithm is bounded by the time required to sort the $2^{\alpha\ell/2}$ pairs (i, x_i); this can be done in time $\mathcal{O}(\ell \cdot 2^{\alpha\ell})$.

Say $c = m^e \bmod N$ with $\|m\| \leq \ell$. If m is chosen as a random ℓ-bit integer, it can be shown that with high 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 $c/r^e = s^e \bmod N$. It is easy to verify that the algorithm finds r, s.

Common modulus attack I. This is a classic example of mis-use 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 ok as long as all employees are trusted (or, alternately, 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) $(e_1 \neq e_2)$. Assume further than $\gcd(e_1, e_2) = 1$. 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, X and Y can be computed efficiently using the extended Euclidean algorithm (see Section B.1.2). We claim that $m = [c_1^X \cdot c_2^Y \bmod N]$ (the latter can be easily calculated); this is true because

$$c_1^X \cdot c_2^Y = m^{Xe_1} m^{Ye_2} = m^{Xe_1 + Ye_2} = m^1 = m \bmod N.$$

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 10.16. The construction is defined based on a parameter ℓ that determines the length of messages that can be encrypted.

CONSTRUCTION 10.16

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:

- *Key-generation algorithm* 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 $sk = \langle N, d \rangle$.
- *Encryption algorithm* 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

  $$c := [\,(r \| m)^e \bmod N].$$

- *Decryption algorithm* 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, note that 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 done in $2^{\mathcal{O}(\log n)} = \mathsf{poly}(n)$ time and the scheme

is completely insecure. 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 different assumption; see Exercise 10.5.) Finally, when $\ell(n) = 1$ it is possible to prove the following:

THEOREM 10.17 *If the RSA problem is hard relative to* GenRSA *then Construction 10.16 with* $\ell(n) = 1$ *has indistinguishable encryptions under chosen-plaintext attacks.*

A full proof[3] of this theorem is beyond the scope of this book; however, one step of the proof is given as part of Exercise 10.6.

PKCS #1 v1.5. A widely-used and standardized encryption scheme, *RSA Laboratories Public-Key Cryptography Standard (PKCS) #1, v1.5*, 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 done 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.)

PKCS #1 v1.5 is believed to be CPA-secure, though 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 that has been proven secure against such attacks. The most recent version of the standard is PKCS #1 v2.1; see Section 13.2.3 for a high-level description of this scheme. This updated version is preferred for new implementations, though the older version is still in widespread use for reasons of backwards-compatibility.

10.5 The El Gamal Encryption Scheme

The El Gamal encryption scheme is another popular and widely-used encryption scheme whose security can be based on the hardness of the decisional

[3] For those who have covered Chapter 6, we note that the theorem relies on the fact that the least-significant bit is hard-core for RSA.

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 analyzing some of the schemes we present in Chapter 10. Let $\mathbb{G}$ be a 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 10.18 *Let $\mathbb{G}$ be a finite group, and let $m \in \mathbb{G}$ be arbitrary. 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 will share as their secret key a random element $g \leftarrow \mathbb{G}$; 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 10.20.

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). The El Gamal encryption scheme is defined as follows:

CONSTRUCTION 10.19

Let $\mathcal{G}$ be as in the text. Define a public-key encryption scheme as follows:

- $\mathsf{Gen}(1^n)$ runs $\mathcal{G}(1^n)$ to obtain $\mathbb{G}, q, g$, and then chooses a random $x \leftarrow \mathbb{Z}_q$. The public key is $\langle \mathbb{G}, q, g, g^x \rangle$ and the private key is $\langle \mathbb{G}, q, g, x \rangle$.
- To encrypt a message $m \in \mathbb{G}$ with respect to the public key $pk = \langle \mathbb{G}, q, g, h \rangle$, choose a random $y \leftarrow \mathbb{Z}_q$ and output the ciphertext
$$\langle g^y,\ h^y \cdot m \rangle.$$
- To decrypt a ciphertext $\langle c_1, c_2 \rangle$ using the private key $sk = \langle \mathbb{G}, q, g, x \rangle$, compute
$$m := c_2 / c_1^x.$$

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 note that

$$\frac{c_2}{c_1^x} = \frac{h^y \cdot m}{(g^y)^x} = \frac{(g^x)^y \cdot m}{g^{xy}} = \frac{g^{xy} \cdot m}{g^{xy}} = m.$$

To fully specify the scheme, we need to show how to encode binary strings as elements of $\mathbb{G}$. See the discussion at the end of this section.

We now prove security of the El Gamal encryption scheme. The reader may want to compare the proof of the following theorem to the proof of Theorem 3.17.

THEOREM 10.20 *If the DDH problem is hard relative to $\mathcal{G}$, then the El Gamal encryption scheme has indistinguishable encryptions under chosen-plaintext attacks.*

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 the experiment depends only on the algorithms for key generation and for encryption.

Lemma 10.18, 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. The first component of the ciphertext is trivially independent of m. Taken together, this means that the distribution of the ciphertext is independent of m and hence 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 $\mathcal{G}$:

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 $\mathsf{Gen}(1^n)$ to obtain $(\mathbb{G}, q, g)$, then choosing random $x, y, z \in \mathbb{Z}_q$, 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 exactly as $\mathcal{A}$'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 $\mathsf{Gen}(1^n)$ to obtain $(\mathbb{G}, q, g)$, then choosing random $x, y \in \mathbb{Z}_q$, 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

$$\begin{aligned}\mathsf{negl}(n) &= \left| \Pr[D(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] - \Pr[D(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1] \right| \\ &= \left| \frac{1}{2} - \varepsilon(n) \right|.\end{aligned}$$

But this implies that $\varepsilon(n) \leq \frac{1}{2} + \mathsf{negl}(n)$, completing the proof. ■

El Gamal in Practice

We briefly discuss a few practical aspects related to El Gamal encryption.

Encrypting 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 $\bar{m}$ is mapped to the quadratic residue $m = [\bar{m}^2 \bmod p\,]$. This encoding is one-to-one and efficiently reversible. It is one-to-one since any quadratic residue $[\bar{m}^2 \bmod p\,]$ has exactly the two square roots $[\pm\bar{m} \bmod p\,]$, and exactly one of these values lies in the range $\{1, \ldots, (p-1)/2\}$. (This is so because $[\bar{m} \bmod p\,] \leq (p-1)/2$ if and only if $[-\bar{m} \bmod p\,] = p - [\bar{m} \bmod p\,] > (p-1)/2$.) The encoding is efficiently invertible since square roots modulo p are easy to compute. See Sections 11.1.1 and 11.2.1 for further discussion about these issues.

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 $\bar{m}$ with $1 \leq \bar{m} \leq q$. Then take $m = [\bar{m}^2 \bmod p\,]$.

Example 10.21

Let $q = 83$ and $p = 2q + 1 = 167$, and let $\mathbb{G}$ be the 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 $\bar{m} = 30$. Squaring this gives $m = [30^2 \bmod 167] = 65 \leq q$. This is our encoding of the message. Picking $y = 71$ when encrypting, we get 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 $\bar{m} = 30$ and $\hat{m}$ can be easily determined. ♢

El Gamal encryption is also widely used with $\mathbb{G}$ being an elliptic-curve group. Such groups were introduced briefly in Section 7.3.4.

Sharing public parameters. Our description of the El Gamal encryption scheme in Construction 10.19 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 the DDH problem is hard relative to $\mathcal{G}$ in the first place, this is because the DDH problem is still believed to be hard even for the party who runs $\mathcal{G}$ to generate $\mathbb{G}, q, g$. We remark that 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 output modulus N. (And in the case of RSA, public parameters can *not* be shared.)

10.6 Chosen-Ciphertext Attacks

Chosen-ciphertext attacks, in which an adversary is allowed to obtain the decryption of any ciphertexts of its choice (with one technical restriction; see the formal definition 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, 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.

As described in Section 4.8, although at first glance it may seem unreasonable to model an attacker as being able to interact with a decryption oracle that decrypts arbitrary ciphertexts of the adversary's choice, there are realistic scenarios in which this model makes sense. 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 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 $\mathcal{R}$.

- $\mathcal{A}$ might also 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 $\mathcal{A}$. Or, $\mathcal{A}$ may not obtain the decryption of c' at all, but the content of m' alone may have beneficial consequences for $\mathcal{A}$. (See the third scenario below for an illustration of this latter point.)

Note that the second class of attacks applies only in the context of public-key encryption. We now give some scenarios demonstrating the above types of attacks.

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 upon a failed login: upon receiving (an encryption of) pw from the 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 and observing the error messages that the bank returns. This information may be enough to enable the adversary to determine the user's password. We remark further that the adversary gains no information by sending c to the bank, since it already knows in this case that no error message will be generated.[4]

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 an encrypted e-mail c' to $\mathcal{R}$ (in its own name), then $\mathcal{R}$ might reply to this e-mail *and quote the decrypted text m' corresponding to c'*. (Note that even if $\mathcal{R}$ encrypts this response using the public key of $\mathcal{A}$, it will be possible for $\mathcal{A}$ to decrypt the response and obtain m'.) 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 $\mathcal{R}$ might get suspicious and refuse to respond.

Scenario 3. Very related to the issue 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 an unknown message m' *that is 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 property further in this section.)

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 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.

We now present a formal definition of security against chosen-ciphertext attacks that exactly parallels Definition 3.31. (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

[4] 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.)

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. *Adversary* $\mathcal{A}$ *is given* pk *and access to a decryption oracle* $\mathsf{Dec}_{sk}(\cdot)$*, outputs a pair of messages* m_0, m_1 *with* $|m_0| = |m_1|$*. (These messages must be in the plaintext space associated with* pk*.)*
3. *A random bit* $b \leftarrow \{0,1\}$ *is chosen, and then the ciphertext* $c \leftarrow \mathsf{Enc}_{pk}(m_b)$ *is computed and given to* $\mathcal{A}$.
4. $\mathcal{A}$ *can continue to interact with the decryption oracle, but may not request 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 10.22 *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*

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

where the probability is taken over the random coins used to generate (pk, sk)*, choose* b*, and encrypt* m_b.

We will not be able to show any examples of CCA-secure encryption schemes here, since 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 the chapter.) In Chapter 13 we discuss a CCA-secure scheme that is used widely in practice. Coming up with simpler or more efficient constructions of CCA-secure public-key encryption schemes, especially based on the RSA assumption, is an area of active research.

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 in the auction scenario discussed earlier (scenario 3). We show how to carry out exactly the attack shown there. 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 $2m$. 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 the 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, the attack is quite practical.

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, but we emphasize again that there is no contradiction since we are now in 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 only negligible probability.) However, this is easy for the adversary to avoid. Letting c_1, c_2, m, m' be as above, $\mathcal{A}$ can now choose a random $y'' \leftarrow \mathbb{Z}_q$ and compute $c_1'' := c_1 \cdot g^{y''}$ and $c_2'' := c_2 \cdot h^{y''} \cdot m'$; then

$$c_1'' = g^y \cdot g^{y''} = g^{y+y''} \quad \text{and} \quad c_2'' = h^y m \cdot h^{y''} m' = h^{y+y''} m m',$$

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 and Public-Key Encryption

We have seen in Section 10.4.3 how to construct a 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 trapdoor permutations and observe that the RSA family of one-way permutations (Construction 7.71) trivially 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. 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 Trapdoor Permutations

We showed in Section 7.4.1 that the RSA assumption naturally gives rise to a family of one-way permutations. But 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 10.23 *A tuple of polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Samp}, f, \mathsf{Inv})$ *is a* family of trapdoor permutations *(or, informally, a* trapdoor permutation*) if the following hold:*

- *The* **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*. It is required that* $(\mathsf{Gen}_1, \mathsf{Samp}, f)$ *be a family of one-way permutations (see Definition 7.70).*
- *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* $x \in \mathcal{D}_I = \mathcal{D}_{\mathsf{td}}$ *we have*

$$\mathsf{Inv}_{\mathsf{td}}(f_I(x)) = x.$$

Informally, the second condition simply means that f_I cannot be efficiently inverted *without* td. The final condition requires that f_I can be inverted *with* 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 $\mathsf{GenModulus}$. We sometimes refer to this as the *RSA trapdoor permutation*. Another example of a trapdoor permutation will be given in Section 11.2.2.

10.7.2 Public-Key Encryption from Trapdoor Permutations

We now sketch, at a somewhat high level, 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 $(\mathsf{Gen}, \mathsf{Samp}, f, \mathsf{Inv})$ 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 in place of f_I, and f_I^{-1} in place of $\mathsf{Inv}_{\mathsf{td}}$, with the understanding that the latter 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-chosen x, we cannot expect to be able to compute x efficiently. Nevertheless, 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*. The following is the natural adaptation of Definition 6.5 to our context.

DEFINITION 10.24 (hard-core predicate): *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 the single bit $\mathsf{hc}_I(x)$.*

We say 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 give (I, td) and then x is chosen uniformly at random from Dec_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 natural way: Given public key I, the sender encrypts a single bit m by (1) choosing 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 $\langle I, \mathsf{td} \rangle$, 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 10.25.) It is easy to see that this recovers the original message.

THEOREM 10.26 *If $\widehat{\Pi}$ is a family of trapdoor permutations and hc is a hard-core predicate of $\widehat{\Pi}$, then Construction 10.25 has indistinguishable encryptions under chosen-ciphertext attacks.*

PROOF Let Π denote the public-key encryption scheme given by Construction 10.25. 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.

CONSTRUCTION 10.25

Let $\widehat{\Pi} = (\widehat{\mathsf{Gen}}, f)$ be a family of permutations, and hc a predicate that is hard-core for $\widehat{\Pi}$. Construct the following public-key encryption scheme:

- *Key-generation algorithm* Gen: On input 1^n, run $\widehat{\mathsf{Gen}}(1^n)$ to obtain (I, td). Output the public key I and the private key $\langle I, \mathsf{td}\rangle$.
- *Encryption algorithm* Enc: On input a public key I and a message $m \in \{0,1\}$, choose random $x \leftarrow \mathcal{D}_I$ and output the ciphertext
$$\langle f_I(x), \mathsf{hc}_I(x) \oplus m\rangle.$$
- *Decryption algorithm* Dec: On input a private key $\langle I, \mathsf{td}\rangle$ and a ciphertext $\langle y, m'\rangle$ with $y \in \mathcal{D}_{\mathsf{td}}$, 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.

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 will 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 $\mathcal{A}_h$ that attempts to compute $h_I(x)$ when given I and $f_I(x)$ as input:

Algorithm $\mathcal{A}_h$:
The algorithm is given I and $y \in \mathcal{D}_I$ as input.

- Set $pk = I$ and run $\mathcal{A}(pk)$ to obtain two messages m_0, m_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 $\bar{z}$, where $\bar{z}$ denotes the complement of z.

Let us analyze the behavior of $\mathcal{A}_h$. Letting x be such that $y = f_I(x)$, we can view z as an initial "guess" by $\mathcal{A}_h$ for the value of $h_I(x)$. This guess is correct with probability $1/2$, and incorrect with probability $1/2$. We also have

$$\begin{aligned}&\Pr[\mathcal{A}_h(I, f_I(x)) = h_I(x)]\\ &= \frac{1}{2}\cdot\Big(\Pr[b' = b \mid z = h_I(x)] + \Pr[b' \neq b \mid z \neq h_I(x)]\Big). \qquad (10.11)\end{aligned}$$

When $z = h_I(x)$ the view of $\mathcal{A}$ (being run as a sub-routine by $\mathcal{A}_h$) is distributed exactly as $\mathcal{A}$'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 $m' =$

$m_b \oplus h_I(x)$, and so the ciphertext $\langle y, m' \rangle$ given to $\mathcal{A}$ is indeed a random encryption of m_b. It follows that

$$\Pr[b' = b \mid z = h_I(x)] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n).$$

On the other hand, when $z \neq h_I(x)$ then the view of $\mathcal{A}$ (being run as a subroutine by $\mathcal{A}_h$) 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

$$m' = m_b \oplus \bar{h}_I(x) = m_{\bar{b}} \oplus h_I(x),$$

(recalling our assumption that $m_0 = \bar{m}_1$), and so the ciphertext $\langle y, m' \rangle$ given to $\mathcal{A}$ is now a random encryption of $m_{\bar{b}}$. Therefore

$$\Pr[b' = b \mid z \neq h_I(x)] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 0] = 1 - \varepsilon(n)$$

and so $\Pr[b' \neq b \mod z \neq h_I(x)] = \varepsilon(n)$.

Using Equation (10.11) and the fact that h is a hard-core predicate for Π, there exists a negligible function negl with

$$\varepsilon(n) = \Pr[\mathcal{A}_h(I, f_I(x)) = h_I(x)] \leq \frac{1}{2} + \mathsf{negl}(n).$$

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 10.27 *If a family of trapdoor permutations* $\widehat{\Pi}$ *exists, then there exists a family of trapdoor permutations* Π *along with a predicate* h *that is hard-core for* Π*.*

Encrypting longer messages. Using Corollary 10.11, we know that we can extend Construction 10.25 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

$$\left\langle f_I(x_1), h_I(x_1) \oplus m_1 \right\rangle, \ldots, \left\langle f_I(x_\ell), h_I(x_\ell) \oplus m_\ell \right\rangle$$

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

$$\langle x_{\ell+1}, h_I(x_1) \oplus m_1, \cdots, h_I(x_\ell) \oplus m_\ell \rangle .$$

A proof that this is secure uses ideas from Section 6.4, and is left as an exercise.

References and Additional Reading

The idea of public-key encryption was first proposed (in the open literature, at least) by Diffie and Hellman [50]. Somewhat amazingly, the El Gamal encryption scheme [60] was not proposed until 1984 even though it can be viewed as a direct transformation of the Diffie-Hellman key exchange protocol shown in Chapter 9; see Exercise 10.3. Rivest, Shamir, and Adleman [110] 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 [70], 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 [27].

The public-key encryption scheme suggested by Rivest, Shamir, and Adleman [110] corresponds to the textbook RSA scheme shown here. The attacks described in Section 10.4.2 are due to [74, 48, 119, 32]; see [93, Chapter 8] and [30] for additional attacks and further explanation. The *PKCS#1 RSA Cryptography Standard* (both the latest version and some previous versions) is available for download from `http://www.rsa.com/rsalabs`. A proof of Theorem 10.17 can be derived from results in [13, 76].

As noted in Chapter 4, chosen-ciphertext attacks were first formally defined by Naor and Yung [99] and Rackoff and Simon [109]. 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 [25]. 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. [51] and Bellare et al. [16]. The first efficient public-key encryption scheme secure against chosen-ciphertext attack was shown by Cramer and Shoup [42]. See the expository article by Shoup [115] for 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 [134], and the efficiency improvement discussed at the end of Section 10.7.2 is due to Blum and Goldwasser [27]. The reader in-

terested in finding out more about hard-core predicates for the RSA family of trapdoor permutations is invited to peruse [13, 76, 12] and references therein.

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 in particular 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 $\mathcal{N}$ possible values. Show how it is possible to determine m in time linear in $\mathcal{N}$.

10.3 Show that any 2-round key-exchange protocol (that is, where each party sends a single message) satisfying Definition 9.2 can be converted into a public-key encryption scheme that is CPA-secure.

10.4 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 (1) always outputs vectors of the same length $t(n)$ for each fixed value of n and such that

$$\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A}',\Pi}(n) = 1].$$

10.5 Let GenRSA have the usual meaning. Consider the following experiment for an algorithm $\mathcal{A}$, a function ℓ with $\ell(n) \leq 2n - 2$ for all n, and a parameter 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 a random bit* b*. If* $b = 0$*, choose random* $y_0 \leftarrow \mathbb{Z}_N^*$. *If* $b = 1$ *choose* $r \leftarrow \{0,1\}^{\|N\|-\ell(n)-1}$ *and set*
>
> $$y_1 := [(r\|m)^e \bmod N].$$
>
> *(d)* $\mathcal{A}$ *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 10.16) using function ℓ has indistinguishable encryptions under chosen-plaintext attacks.

10.6 We say a function f is *hard-core for* GenRSA if for all probabilistic, polynomial-time algorithms $\mathcal{A}$ there exists a negligible function negl such that

$$\Big|\Pr[\mathcal{A}(N, e, c, f(x)) = 1] - \Pr[\mathcal{A}(N, e, c, f(r)) = 1]\Big| \leq \mathsf{negl}(n),$$

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 c 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 [12]. Prove Theorem 10.17 by relying on this result.

Hint: Note that in Construction 10.16, $\mathsf{msb}(r\|m)$ is always equal to 0.

10.7 Consider the following variant of Construction 10.25:

CONSTRUCTION 10.28

Let $\widehat{\Pi} = (\widehat{\mathsf{Gen}}, f)$ and h be as in Construction 10.25.

- *Key-generation algorithm* Gen: As in Construction 10.25.
- *Encryption algorithm* Enc: On input a public key I and a message $m \in \{0,1\}$, choose random $x \leftarrow \mathcal{D}_I$ such that $h_I(x) = m$, and output the ciphertext $f_I(x)$.
- *Decryption algorithm* Dec: On input a private key $\langle I, \mathsf{td}\rangle$ and a ciphertext y with $y \in \mathcal{D}_{\mathsf{td}}$, compute $x := f_I^{-1}(y)$ and output the message $h_I(x)$.

(a) Argue that encryption can be performed in polynomial time.

(b) Prove that if $\widehat{\Pi}$ is a family of trapdoor permutations and h is a hard-core predicate of $\widehat{\Pi}$, then this construction has indistinguishable encryptions under chosen-ciphertext attacks.

Chapter 11

* Additional Public-Key Encryption Schemes

In the previous chapter we have seen 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. The schemes discussed in this chapter were not placed here because they are fundamentally 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 in this chapter require a bit more number theory than we have covered to this point. Actually, each of the schemes we show here is well worth understanding, from at least 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 satisfy Definition 10.3. The scheme is relatively simple to describe, and the cryptographic assumption on which it relies is 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. It is more efficient than the Goldwasser-Micali or Rabin cryptosystems, as well as the provably-secure RSA scheme of Theorem 10.17. The Paillier encryption scheme also has other advantages 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.

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 $x \in \mathbb{G}$ such that $x^2 = y$. An element that is not a quadratic residue is called a *quadratic non-residue*. It is not too hard to show that in an abelian group, the set of quadratic residues form a subgroup.

In the specific case of $\mathbb{Z}_p^*$, we have that y is a quadratic residue if there exists an x with $y = x^2 \bmod p$. 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 $p \,|\, 2x$. However, this means 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 distinct square roots.

Let $x' \in \mathbb{Z}_p^*$ be a square root of y. Then $x^2 = (x')^2 \bmod p$ 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

$$|\mathcal{QR}_p| = |\mathcal{QNR}_p| = \left|\mathbb{Z}_p^*\right|/2 = (p-1)/2\,.$$

Define[1] $\mathcal{J}_p(x)$, the *Jacobi symbol of x modulo p*, as follows. 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 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

$$x^{\frac{p-1}{2}} = \left(g^{2j+1}\right)^{\frac{p-1}{2}} = \left(g^{2j}\right)^{\frac{p-1}{2}} \cdot g^{\frac{p-1}{2}} = 1 \cdot g^{\frac{p-1}{2}} = g^{\frac{p-1}{2}} \bmod p.$$

Now,

$$\left(g^{\frac{p-1}{2}}\right)^2 = g^{p-1} = 1 \bmod p,$$

[1] $\mathcal{J}_p(x)$ is also sometimes called the *Legendre symbol* of x, and denoted by $\mathcal{L}_p(x)$; we have chosen alternate notation to be consistent with notation introduced later.

and so $g^{\frac{p-1}{2}} = \pm 1 \bmod p$ since $[\pm 1 \bmod p]$ are the two square roots of 1. Since g is a generator, it has order $p-1$ and so $g^{\frac{p-1}{2}} \neq 1$. It follows that $x^{\frac{p-1}{2}} = -1 = \mathcal{J}_p(x) \bmod p$ in this case as well. ■

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: Prime p; element $x \in \mathbb{Z}_p^*$
Output: $\mathcal{J}_p(x)$

$b := [x^{\frac{p-1}{2}} \bmod p]$
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 a 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 let $y \in \mathbb{Z}_N^*$ 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 $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$. That is,

$$y_p = x_p^2 \bmod p \quad \text{and} \quad y_q = x_q^2 \bmod q \tag{11.1}$$

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. ■

The above proposition characterizes the quadratic residues modulo N. Careful examination of the proof yields more: in fact, 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 with square root 2. Since $2 \leftrightarrow (2,2)$, the other square roots of 4 are given by

- $\big(2, [-2 \bmod 3]\big) = (2,1) \leftrightarrow 3$;
- $\big([-2 \bmod 5], 2\big) = (3,2) \leftrightarrow 8$; and
- $\big([-2 \bmod 5], [-2 \bmod 3]\big) = (3,1) \leftrightarrow 13$.

One can verify that $7^2 = 8^2 = 13^2 = 4 \bmod 15$. ♢

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,

$$\begin{aligned}\mathcal{J}_N(x) &\stackrel{\text{def}}{=} \mathcal{J}_p(x) \cdot \mathcal{J}_q(x) \\ &= \mathcal{J}_p([x \bmod p\,]) \cdot \mathcal{J}_q([x \bmod q\,]).\end{aligned}$$

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

$$\mathcal{QNR}_N^{+1} \stackrel{\text{def}}{=} \left\{ x \in \mathbb{Z}_N^* \;\middle|\; \begin{array}{c} \mathcal{J}_n(x) = +1, \text{ but } x \text{ is not} \\ \text{a quadratic residue modulo } N \end{array} \right\}.$$

It is now easy to prove the following:

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 say $x, y \in \mathbb{Z}_N^*$. 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 $xy \in \mathcal{QNR}_N^{+1}$. ■

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.*

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 immediate way of efficiently computing the Jacobi symbol modulo N, or efficiently deciding whether a given element x is a quadratic residue modulo N or not. Somewhat surprisingly, a highly non-trivial 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.) 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 a reasonable cryptographic assumption that *no* polynomial-time algorithm for deciding quadratic residuosity in this case (that performs better than random guessing) exists.

We now formalize this 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.

DEFINITION 11.11 *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*

$$\Big|\Pr[\mathcal{A}(N, \mathsf{qr}) = 1] - \Pr[\mathcal{A}(N, \mathsf{nqr}) = 1]\Big| \leq \mathsf{negl}(n),$$

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 $\mathcal{QNR}_N^{+1}$.*

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 will be a modulus N, and the secret key will be the factorization of N.
- The encryption of the bit '0' will be a random quadratic residue, and the encryption of the bit '1' will be 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.

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 $\mathcal{QNR}_N^{+1}$ (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 Section B.2.4) and set $y := x^2 \bmod N$. Clearly $y \in \mathcal{QR}_N$. That y is random follows from the facts that squaring modulo N is a 4-to-1 function (see Section 11.1.2) and x is chosen at random. In more detail, fix any $\hat{y} \in \mathcal{QR}_N$ and let us compute the probability that $y = \hat{y}$. 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 can help.* Specifically, we modify the scheme as described above so that the receiver will additionally choose a random $z \leftarrow \mathcal{QNR}_N^{+1}$ and include 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.) Then the sender can choose a random element $y \leftarrow \mathcal{QNR}_N^{+1}$ by first choosing a random $x \leftarrow \mathcal{QR}_N$ (as above) and then setting $y := [z \cdot x \bmod N]$. We leave it as an exercise to show that y chosen in this way is indeed 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, as Construction 11.12.

THEOREM 11.13 *If the quadratic residuosity problem is hard relative to* GenModulus*, then the Goldwasser-Micali encryption scheme has indistinguishable encryptions under chosen-plaintext attacks.*

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 GenPrime:

CONSTRUCTION 11.12

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. Construct a public-key encryption scheme as follows:

- $\mathsf{Gen}(1^n)$ runs $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q). It also chooses random $z \leftarrow \mathcal{QNR}_N^{+1}$. The public key is $pk = \langle N, z \rangle$ and the private key is $sk = \langle p, q \rangle$.
- To encrypt a message $m \in \{0, 1\}$ with respect to the public key $pk = \langle N, z \rangle$, choose random $x \leftarrow \mathbb{Z}_N^*$ and output the ciphertext

$$c := [z^m \cdot x^2 \bmod N].$$

- To decrypt a ciphertext c using the private key $sk = \langle p, q \rangle$, determine whether c is a quadratic residue modulo N using, e.g., the algorithm of Section 11.1.3. If c is a quadratic residue, output 0; otherwise, output 1.

The Goldwasser-Micali encryption scheme.

Algorithm D:
The algorithm is given N, z as input.

- Set $pk = \langle N, z \rangle$ and run $\mathcal{A}(pk)$ to obtain two messages m_0, m_1.
- Choose a random bit b and a random $x \leftarrow \mathbb{Z}_N^*$, and 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 random $z \leftarrow \mathcal{QNR}_N^{+1}$. 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 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

$$\Pr[D(N, \mathsf{qr}) = 1] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)\,,$$

where qr represents a random quadratic residue as in Definition 11.11.

Case 2: Say the input to D was generated by running $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q), and then choosing 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 it is immediate that 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 10.18 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,

$$\Pr[D(N, \mathsf{qnr}) = 1] = \frac{1}{2},$$

where qnr represents a random element of $\mathcal{QNR}_N^{+1}$ as in Definition 11.11.

Since the quadratic residuosity problem is hard relative to GenModulus, there must exist a negligible function negl such that

$$\begin{aligned}\mathsf{negl}(n) &= \left|\Pr[D(N, \mathsf{qr}) = 1] - \Pr[D(N, \mathsf{qnr}) = 1]\right| \\ &= \left|\varepsilon(n) - \frac{1}{2}\right|.\end{aligned}$$

This implies that $\varepsilon(n) \leq \frac{1}{2} + \mathsf{negl}(n)$, completing the proof. ■

11.2 The Rabin Encryption Scheme

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 to be 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 factoring N; thus, the factoring assumption implies the difficulty of computing square roots modulo a composite (generated appropriately). Due of this equivalence, a version of the Rabin encryption scheme can be shown to be CPA-secure as long as factoring is hard. Such a result is *not* known for RSA encryption, and the RSA problem may potentially be easier than factoring. (A similar remark applies to the Goldwasser-Micali encryption scheme.) This makes the Rabin encryption scheme very attractive, at least from a theoretical point of view.

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 the latter is more widely-used than the former seems to be due to historical factors rather than technical ones; we discuss these issues 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 turns out to be relatively simple when $p = 3 \bmod 4$, and much more involved when $p = 1 \bmod 4$. We tackle the easier case first. (Actually, 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 $x := [a^{\frac{p+1}{4}} \bmod p]$.

Note that it is crucial, above, that $(p+1)/2$ is *even*, and so $(p+1)/4$ is an *integer* (this is necessary in order for it to be possible to efficiently compute $a^{\frac{p+1}{4}} \bmod p$). This approach does not succeed when $p = 1 \bmod 4$, in which case $p + 1$ is an integer that is *not* divisible by 4.

Instead, when $p = 1 \bmod 4$ we will proceed slightly differently. Motivated by the above approach, we might think to search for an *odd* integer r such 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 11.14
Take $p = 29$ and $a = 22$. Then

$$22^7 \cdot 2^{14} = 1 \bmod 29,$$

and so $15 = 22^{(7+1)/2} \cdot 2^{14/2} = 22^4 \cdot 2^7 \bmod 29$ 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

$$a^{2^\ell m} = a^{\frac{p-1}{2}} = 1 \bmod p. \tag{11.3}$$

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 has been reduced by a factor of 2, and more importantly is 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 11.15

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 11.15. 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

$$a^{2^{\ell'} m} \cdot b^{2^\ell m} = (-1)(-1) = +1 \bmod p.$$

[2] The integers ℓ and m can be computed easily by taking out factors of 2 from $(p-1)/2$.

ALGORITHM 11.16
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 is odd, r' is even, and $a^m \cdot b^{r'} = 1 \bmod p$ */
  return $\left[a^{\frac{r+1}{2}} \cdot b^{\frac{r'}{2}} \bmod p\right]$

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 above. 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 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). We remark that 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 11.17
In this example, 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 11.18

Here we work out a concrete example. Let $p = 17$ (so $(p-1)/2 = 2^3$), $a = 4$, and $b = 3$. Note that here $m = 1$.

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. That is:

- Compute $a_p := [a \bmod p]$ and $a_q := [a \bmod q]$.
- Using the algorithm just shown, 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.

11.2.2 A Trapdoor Permutation based on Factoring

We have seen that computing square roots modulo N can be done in polynomial time if the factorization of N is known. We show here that 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 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 11.19 *We say* 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 is 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 11.20 *Let $N = pq$ with p, q distinct, odd primes. Given x_1, x_2 such that $x_1^2 = y = x_2^2 \bmod N$ but $x_1 \neq \pm x_2 \bmod N$, it is possible to factor N in time polynomial in $\|N\|$.*

PROOF We claim that either $\gcd(N, x_1 + x_2)$ or $\gcd(N, x_1 - x_2)$ is equal to one of the prime factors of N.[3] Since gcd computations can be carried out in polynomial time, this proves the lemma.

If $x_1^2 = x_2^2 \bmod N$ then

$$0 = x_1^2 - x_2^2 = (x_1 - x_2) \cdot (x_1 + x_2) \bmod N,$$

and so $N \mid (x_1 - x_2)(x_1 + x_2)$. Then $p \mid (x_1 - x_2)(x_1 + x_2)$ and so p divides one of these terms. Say $p \mid (x_1 + x_2)$ (the proof proceeds similarly if $p \mid (x_1 - x_2)$). If $q \mid (x_1 + x_2)$ then $N \mid (x_1 + x_2)$, but this cannot be the case since $x_1 \neq -x_2 \bmod N$. So $q \not| x + 1 + x_2$ and $\gcd(N, x_1 + x_2) = p$. ■

An alternate way of proving the above is to look at what is going on in the Chinese remaindering representation. Say $x_1 \leftrightarrow (x_{1,p}, x_{1,q})$. Then, because x_1 and x_2 are square roots of the same value y, we know that x_2 corresponds to either $(-x_{1,p}, x_{1,q})$ or $(x_{1,p}, -x_{1,q})$. (It cannot correspond to $(x_{1,p}, x_{1,q})$ or $(-x_{1,p}, -x_{1,q})$ since the first corresponds to x_1 while the second corresponds to $-x_1 \bmod N$, and both possibilities are ruled out by assumption.) Say $x_2 \leftrightarrow (-x_{1,p}, x_{1,q})$. Then

$$[x_1 + x_2 \bmod N] \leftrightarrow (x_{1,p}, x_{1,q}) + (-x_{1,p}, x_{1,q}) = (0, [2x_{1,q} \bmod q]),$$

and we see that $x_1 + x_2 = 0 \bmod p$ while $x_1 + x_2 \neq 0 \bmod q$. Since $x_1 \neq x_2$, we have that $\gcd(N, x_1 + x_2) = p$, a factor of N.

We can now prove the main result of this section.

THEOREM 11.21 *If factoring is hard relative to* GenModulus*, then computing square roots is hard relative to* GenModulus *as well.*

PROOF Let $\mathcal{A}$ be a probabilistic, polynomial-time algorithm, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr\left[\mathsf{SQR}_{\mathcal{A},\mathsf{GenModulus}}(n) = 1\right].$$

[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.

Consider the following probabilistic, polynomial-time algorithm $\mathcal{A}_{\text{fact}}$ that attempts to factor moduli output by $\mathsf{GenModulus}$:

Algorithm $\mathcal{A}_{\text{fact}}$:
The algorithm is given a modulus N as input.

- Choose random $x_1 \leftarrow \mathbb{Z}_N^*$ and compute $y := x_1^2 \bmod N$.
- Run $\mathcal{A}(N, y)$ to obtain output x_2.
- If $x_2^2 = y \bmod N$ and $x_2 \neq \pm x_1 \bmod N$, then factor N.

(In the third step, we rely on Lemma 11.20.)

By Lemma 11.20, we know that $\mathcal{A}_{\text{fact}}$ succeeds in factoring N exactly when $x_2 \neq \pm x_1 \bmod N$ and $x_2^2 = y \bmod N$. That is,

$$\begin{aligned}
&\Pr[\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenModulus}}(n) = 1] \\
&= \Pr\left[x_2 \neq \pm x_1 \bmod N \bigwedge x_2^2 = y \bmod N\right] \\
&= \Pr\left[x_2 \neq \pm x_1 \bmod N \,\middle|\, x_2^2 = y \bmod N\right] \cdot \Pr\left[x_2^2 = y \bmod N\right], \quad (11.4)
\end{aligned}$$

where the above probabilities all refer to experiment $\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenPrime}}(n)$. In this 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 (see Section 11.1.4 if this is unclear). 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,

$$\Pr\left[x_2^2 = y \bmod N\right] = \Pr\left[\mathsf{SQR}_{\mathcal{A},\mathsf{GenModulus}}(n) = 1\right] = \varepsilon(n). \quad (11.5)$$

Conditioned on the value of the quadratic residue y chosen in any given run of experiment $\mathsf{Factor}_{\mathcal{A}_{\text{fact}},\mathsf{GenPrime}}(n)$, the value x_1 is equally likely to be each of the four possible square roots of y. From the point of view of algorithm $\mathcal{A}$ (being run as a subroutine by $\mathcal{A}_{\text{fact}}$), then, x_1 is equally likely to be each of the four square roots of y. This in turn means that, conditioned on $\mathcal{A}$'s outputting *some* square root x_2 of y, the probability that $x_2 = \pm x_1 \bmod N$ is exactly $1/2$. (We stress that we do not make any assumption about how x_2 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_1 is uniformly distributed among the square roots of y from the perspective of $\mathcal{A}$.) That is,

$$\Pr\left[x_2 \neq \pm x_1 \bmod N \,\middle|\, x_2^2 = y \bmod N\right] = \frac{1}{2}. \quad (11.6)$$

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 GenModulus, there must exist a negligible function negl such that

$$\mathsf{negl}(n) \geq \Pr[\mathsf{Factor}_{\mathcal{A}_{\mathrm{fact}},\mathsf{GenModulus}}(n) = 1] = \frac{1}{2} \cdot \varepsilon(n),$$

completing the proof. ■

The previous theorem leads directly to a family of one-way functions (see Definition 7.70) based on any GenModulus relative to which factoring is hard:

- Algorithm Gen, on input 1^n, runs $\mathsf{Gen}(1^n)$ to obtain (N, p, q) and outputs $I = N$. The domain $\mathcal{D}_I$ will be $\mathbb{Z}_N^*$ and the range $\mathcal{R}_I$ will be $\mathcal{QR}_N$.
- 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 $[x^2 \bmod N]$.

We can turn this into a *permutation* by using moduli N of a special form and working over a subset of $\mathbb{Z}_N^*$. Say $N = pq$ is a *Blum integer* if p and q are distinct primes with $p = q = 3 \bmod 4$. The key is the following proposition.

PROPOSITION 11.22 *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. 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 $\mathcal{J}_p(x_q) = -1$ (the proof proceeds similarly in any other case). Using Proposition 11.4, we have

$$\mathcal{J}_p(-x_p) = \mathcal{J}_p(-1) \cdot \mathcal{J}_p(x_p) = -1$$

and, similarly, $\mathcal{J}_p(-x_q) = 1$.

The above shows that x_p and $-x_q$ are quadratic residues modulo p and q, respectively, and that $-x_p$ and x_q are *not* quadratic residues modulo p and q, respectively. Using the characterization of quadratic residues modulo N given by Proposition 11.6, we see that $(x_p, -x_q)$ is a quadratic residue modulo N, but none of the other square roots of y are. ■

Expressed differently, the above proposition says that when N is a Blum integer, the function $f_N : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$ 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 $\mathsf{GenModulus}$ relative to which factoring is hard. (This is sometimes called the *Rabin* family of trapdoor permutations.) In summary:

THEOREM 11.23 *Let* $\mathsf{GenModulus}$ *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) and* $p = q = 3 \bmod 4$*. If factoring is hard relative to* $\mathsf{GenModulus}$*, then there exists a family of trapdoor permutations.*

11.2.3 The Rabin Encryption Scheme

In Section 10.7.2 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 10.24). It can be shown that the least-significant bit $\mathsf{lsb}(\cdot)$ constitutes a hard-core predicate. That is, let $\mathsf{GenModulus}$ 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 $\mathsf{Gen}(1^n)$ outputs (N, p, q) and then x is chosen at random from $\mathcal{QR}_N$. Plugging this into Construction 10.25 gives a concrete example of a public-key encryption scheme whose security can be based on the factoring assumption.

An alternate approach, perhaps conceptually simpler though less efficient than the above, is to construct a "padded Rabin" encryption scheme by analogy with padded RSA encryption (see Section 10.4.3). In Construction 11.24 we describe such an approach for encrypting single-bit messages. See Exercise 11.11 for a generalization to longer messages (which has no known proof of security based on factoring).

We do not prove the following theorem here; see Exercise 10.6 of Chapter 10 (and the reference there) for an idea as to how a proof would proceed.

THEOREM 11.25 *If factoring is hard relative to* $\mathsf{GenModulus}$*, the Construction 11.24 has indistinguishable encryptions under chosen-plaintext attacks.*

CONSTRUCTION 11.24

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) with $p = q = 3 \bmod 4$. Construct a public-key encryption scheme as follows:

- $\mathsf{Gen}(1^n)$ runs $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q). The public key is N, and the private key is $\langle p, q \rangle$.
- To encrypt a message $m \in \{0,1\}$ with respect to the public key N, repeatedly choose random $r \in \{0,1\}^{\|N\|-2}$ until $r \| m$ (viewed as an element of $\mathbb{Z}_N^*$ in the natural way) is a quadratic residue modulo N. Output the ciphertext

$$c := [(r\|m)^2 \bmod N].$$

- To decrypt a ciphertext $c \in \mathcal{QR}_N$ using the private key $\langle p, q \rangle$, compute the unique $\hat{r} \in \mathcal{QR}_N$ such that $\hat{r}^2 = c \bmod N$, and output the least-significant bit of $\hat{r}$.

The padded Rabin encryption scheme.

Rabin Encryption vs. RSA Encryption

It is worthwhile to remark on the similarities and differences between the Rabin and RSA cryptosystems. For concreteness, the reader can think in terms of Construction 10.16 (with $\ell = 1$) and Construction 11.24, though the discussion here applies more generally to any 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, in some sense, based on a weaker assumption; it is theoretically possible that someone might one day develop an efficient algorithm for solving the RSA problem, yet computing square roots will remain hard. More plausible is that an algorithm will one day be proposed that solves the RSA problem in less time than it takes to factor (but still requiring super-polynomial time); computing square roots modulo N, however, can never be much faster than the best available algorithm for factoring N.

Efficiency-wise, the RSA and Rabin permutations are essentially the same. Actually, if a large exponent e is chosen in the case of RSA then computation in the "easy" direction is slightly slower with RSA than with 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 11.9). No such attack is known in the case of textbook RSA.

The RSA permutation is much more widely used in practice than the Rabin permutation, and 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 10.17 and 11.25, respectively. Perhaps more importantly, the Paillier encryption scheme possesses some nice *homomorphic* properties we will discuss further in Section 11.3.3.

The Paillier encryption 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[4] $\mathbb{Z}_N \times \mathbb{Z}_N^*$ (see Definition 7.23) for N of the form we will be interested in.

PROPOSITION 11.26 *Let $N = pq$, where p, q are distinct odd primes of the same length. Then:*

1. $\gcd(N, \phi(N)) = 1$.
2. *For a an integer with $0 \le a \le N$, 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 $(1+N)^x \neq 1 \bmod N^2$ for any $1 \le x < N$.*
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*
$$f(a, b) = [(1+N)^a \cdot b^N \bmod N^2].$$

[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 as in 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).

Section 11.3.1 is dedicated to a proof of Proposition 11.26. The reader who is willing to accept it 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 11.26 claim-by-claim. Throughout, we let N, p, q be as in the proposition.

CLAIM 11.27 *For N, p, q as in Proposition 11.26,* $\gcd(N, \phi(N)) = 1$.

PROOF Recall $\phi(N) = (p-1)(q-1)$. 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 11.28 *For a an integer with $0 \leq a \leq N$, 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), we see that

$$(1+N)^a = \sum_{i=0}^{a} \binom{a}{i} N^i.$$

Reducing the right-hand side modulo N^2, all terms with $i \geq 2$ become 0 and so $(1+N)^a = 1 + aN \bmod N^2$. For a in the range $\{1, \ldots, N\}$, this expression is equivalent to 1 modulo N^2 only when $a = N$. ■

CLAIM 11.29 *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 $b_2 \in \mathbb{Z}_{N^2}^*$ and so it 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 11.28, $(1+N)$ 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 11.27, 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 is taking place modulo N^2, while addition/multiplication on the right-hand side is taking 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 11.28), 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 $1 \leq r \leq N-1$ (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 11.28. Plugging this in to Equation (11.8) we get the desired result:

$$f(a_1, b_1) \cdot f(a_2, b_2) = (1+N)^{a_1 + a_2 \bmod N} \cdot (b_1 b_2 \bmod N)^N \bmod N^2,$$

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 11.26 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, (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 N^{th} *residue modulo* N^2 if y is an N^{th} power; that is, if there exists an $x \in \mathbb{Z}_{N^2}^*$ with $y = x^N \bmod N^2$. Let us characterize the N^{th} residues in $\mathbb{Z}_{N^2}^*$. Taking any element $x \in \mathbb{Z}_{N^2}^*$ with $x \leftrightarrow (a, b)$ and raising it to the N^{th} 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 N^{th} 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 the set of N^{th} residues corresponds exactly to the set

$$\mathsf{Res}(N^2) \stackrel{\text{def}}{=} \{(0, b) \mid b \in \mathbb{Z}_N^*\}\,.$$

(Compare this to $\mathbb{Z}_{N^2}^*$, which corresponds to $\{(a, b) \mid a \in \mathbb{Z}_N,\ b \in \mathbb{Z}_N^*\}$.)

The *decisional composite residuosity problem*, roughly speaking, is to distinguish a random element of $\mathbb{Z}_{N^2}^*$ from a random element of $\mathsf{Res}(N^2)$. Formally, 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). Then:

DEFINITION 11.30 *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 in the first case, $[r^N \bmod N^2]$ is a random element of* $\mathsf{Res}(N^2)$.)

The *decisional composite residuosity (DCR) assumption* is simply 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 saw 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 N^{th} 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 N^{th} residue $(0, r)$ and set the ciphertext equal to

$$c := (m, 1) \cdot (0, r) = (m, r).$$

(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 N^{th} 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) from a ciphertext constructed as

$$c' := (m, 1) \cdot (r', r) = ([m + r' \bmod N], r)$$

for random $r' \in \mathbb{Z}_N$. Lemma 10.18 shows that $[m + r' \bmod N]$ is uniformly distributed in $\mathbb{Z}_N$ and so, in particular, the 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 will take place in the isomorphic group $\mathbb{Z}_{N^2}^*$. That is, the sender will generate a ciphertext $C \in \mathbb{Z}_{N^2}^*$ by choosing random $r \in \mathbb{Z}_N^*$ and then computing

$$\begin{aligned} c &:= [(1+N)^m \cdot r^N \bmod N^2] \\ &= \left((1+N)^m \cdot 1^N\right) \cdot \left((1+N)^0 \cdot r^N\right) \bmod N^2. \end{aligned}$$

Note that $c \leftrightarrow (m, r)$ 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 each case the distribution on $r^N \bmod N^2$ 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 $m := \hat{m} \cdot \phi(N)^{-1} \bmod N$.

To see why this works, let $c \leftrightarrow (m, r)$ for arbitrary $r \in \mathbb{Z}_N^*$. Then

$$\begin{aligned} \hat{c} &\stackrel{\text{def}}{=} [c^{\phi(N)} \bmod N^2] \\ &\leftrightarrow (m,\, r)^{\phi(N)} \\ &= \Big([m \cdot \phi(N) \bmod N],\ [r^{\phi(N)} \bmod N]\Big) \\ &= \left([m \cdot \phi(N) \bmod N],\ 1\right). \end{aligned}$$

By Proposition 11.26(3), this means that $\hat{c} = (1+N)^{[m \cdot \phi(N) \bmod N]} \bmod N^2$. Using Proposition 11.26(2), we know that

$$\hat{c} = (1+N)^{[m \cdot \phi(N) \bmod N]} = (1 + [m \cdot \phi(N) \bmod N] \cdot N) \bmod N^2,$$

and so $\hat{m} \stackrel{\text{def}}{=} (\hat{c} - 1)/N = [m \cdot \phi(N) \bmod N]$. Finally,

$$m \stackrel{\text{def}}{=} [\hat{m} \cdot \phi(N)^{-1} \bmod N] = m,$$

as required. (Note that $\phi(N)$ is invertible modulo N since $\gcd(N, \phi(N)) = 1$.)

We give an example of the above calculations, followed by a complete description of the Paillier encryption scheme.

Example 11.31
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

$$\hat{m} := (23911 - 1)/187 = 137;$$

since $90 = [160^{-1} \bmod 187]$, the message is recovered as

$$m := [137 \cdot 90 \bmod 187] = 175.$$

♢

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:

- $\mathsf{Gen}(1^n)$ runs $\mathsf{GenModulus}(1^n)$ to obtain (N, p, q). The public key is N, and the private key is $\langle N, \phi(N)\rangle$.
- To encrypt a message $m \in \mathbb{Z}_N$ with respect to the public key N, choose random $r \leftarrow \mathbb{Z}^*_N$ and output the ciphertext

$$c := [(1 + N)^m \cdot r^N \bmod N^2].$$

- To decrypt a ciphertext c using the private key $sk = \langle N, \phi(N)\rangle$, compute

$$m := \left[\frac{[c^{\phi(N)} \bmod N^2] - 1}{N} \cdot \phi(N)^{-1} \bmod N \right].$$

The Paillier encryption scheme.

Correctness follows from what we have shown earlier. We now prove security of the scheme.

THEOREM 11.33 *If the decisional composite residuosity problem is hard relative to* GenModulus*, then the Paillier encryption scheme has indistinguishable encryptions under chosen-plaintext attacks.*

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 $\mathsf{GenModulus}$:

> **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 $y := [r^N \bmod N^2]$. In this case, the ciphertext c is constructed as

$$c = [(1+N)^{m_b} \cdot r^N \bmod N^2]$$

for 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

$$\Pr[D(N, r^N) = 1] = \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] = \varepsilon(n)\,,$$

where the left-most probability is taken over the appropriate experiment indicated in Definition 11.30.

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 10.18) 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 indicated in Definition 11.30.

Since, by assumption, the decisional composite residuosity problem is hard relative to GenModulus, there must exist a negligible function negl such that

$$\begin{aligned}\mathsf{negl}(n) &= \left|\Pr[D(N,[r^N \bmod N^2])=1] - \Pr[D(N,r)=1]\right| \\ &= \left|\varepsilon(n) - \frac{1}{2}\right|.\end{aligned}$$

This implies that $\varepsilon(n) \leq \frac{1}{2} + \mathsf{negl}(n)$, 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

$$\begin{aligned}&\left((1+N)^{m_1} \cdot r_1^N\right) \cdot \left((1+N)^{m_2} \cdot r_2^N\right) \\ &\quad = (1+N)^{[m_1+m_2 \bmod N]} \cdot (r_1 r_2)^N \bmod N^2,\end{aligned}$$

and the latter is a valid encryption of the message $[m_1 + m_2 \bmod N]$.

DEFINITION 11.34 *A public-key encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* homomorphic *if for all n and all (pk, sk) output by $\mathsf{Gen}(1^n)$, it is possible to define groups $\mathbb{C}, \mathbb{M}$ 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.

Re-stating what we have said above, the Paillier encryption scheme is homomorphic taking $\mathbb{C} = \mathbb{Z}_{N^2}^*$ and $\mathbb{M} = \mathbb{Z}_N$ 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 ciphertexts are elements of $\mathbb{G} \times \mathbb{G}$ and messages are elements of $\mathbb{G}$. Furthermore,

$$\langle g^{y_1},\ h^{y_1} \cdot m_1 \rangle \cdot \langle g^{y_2},\ h^{y_2} \cdot m_2 \rangle = \langle g^{y_1+y_2},\ h^{y_1+y_2} \cdot m_1 m_2 \rangle,$$

a valid encryption of the message $m_1 m_2 \in \mathbb{G}$. Goldwasser-Micali encryption is also homomorphic (see Exercise 11.12).

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" vote, and let 1 stand for a "yes" vote. 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 randomly-chosen $r \leftarrow \mathbb{Z}_N^*$.
3. Each voter broadcasts their vote c_i. These votes are then *aggregated* by computing
$$c^* := \left[\prod_{i=1}^{\ell} c_i \bmod N^2 \right].$$
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
$$v^* = \sum_{i=1}^{\ell} v_i.$$

Note that the authority obtains the correct vote total *without learning any individual votes*. Furthermore, *no voter learns anyone else's vote, either*. We remark that we assume here that all parties act honestly (and only try to figure out 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

The books by Childs [36] and Shoup [117] provide further coverage of the number theory (and 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 [49]. The problem of deciding quadratic residuosity modulo a composite of unknown factorization goes back to Gauss [61] and is related to other (conjectured) hard number-theoretic problems. The Goldwasser-Micali encryption scheme is from [70], and was the first public-key encryption scheme with a rigorous proof of security.

Rabin [107] showed that computing square roots modulo a composite is 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 [26].

Section 10.2 of Shoup's text [117] 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 [105].

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 $\mathbb{Z}_N$. (I.e., 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 $\mathcal{QNR}_N^{+1}$ *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 $[x^{-1} \bmod N] \in \mathcal{QR}_N$. Similarly, prove that if $x \in \mathcal{QNR}_N^{+1}$ then $[x^{-1} \bmod N] \in \mathcal{QNR}_N^{+1}$.

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 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 the message $m \in \{0, 1\}$, the sender chooses random $x \in \mathbb{Z}_N$ and computes the ciphertext $c := (-1)^m \cdot x^2 \bmod N$. Decryption is done as in Construction 11.12.

(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 chosen-plaintext attacks if deciding quadratic residuosity is hard relative to GenModulus.

Hint: The proof is slightly easier than the proof of Theorem 11.13.

11.7 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) State 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 GenModulus, then this scheme has indistinguishable encryptions under chosen-plaintext attacks.

Hint: Use Corollary 11.10.

11.8 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.

11.9 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 encryption scheme that recovers the entire private key.

11.10 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.

11.11 (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) Using the previous result, suggest a variant of the 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.)

11.12 Show that the Goldwasser-Micali encryption scheme is homomorphic, where the plaintext space $\{0,1\}$ is viewed as the group $\mathbb{Z}_2$.

Chapter 12

Digital Signature Schemes

12.1 Digital Signatures – An Overview

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. Signature schemes can be viewed as the public-key counterpart of message authentication codes, though there are some important differences as we will see below.

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 this 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; the pair (m, σ) is then sent 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. Note that 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. m' might be a modified version of some previous patch m, or it might be completely new and unrelated to previous patches. If the signature scheme is "secure" (in a sense we will define more carefully soon), however, then when the client attempts to verify σ' it will find that this is an *invalid* signature on m' with respect to pk; that is, the client will *reject* the signature. Note that the client should reject even if m' is modified only slightly (even only in

a single bit!) from a genuine patch m.

Comparison to Message Authentication Codes

Both message authentication codes and digital signature schemes are used to ensure the integrity (or authenticity) of transmitted messages. Using digital signatures rather than message authentication codes simplifies key management in the case 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 message authentication code with respect to each such key; instead, the sender need only compute only 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 verify it as legitimate, too. This feature is not achieved by message authentication codes when 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 12.7.

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 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

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. Finally, even if we assume that 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 S generated tag rather than R himself — the very fact that message authentication codes are *symmetric* (so that anything S can do, R can also do) implies that there is no way for the judge to distinguish between 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 parties are able to share a secret key in advance of their communication, message authentication codes may be preferable. We remark that there may also be settings where non-repudiation and transferability are specifically *not* desirable: say, when a signer S wants a *particular* recipient to be assured that S certified a message, but does not want this recipient to be able to prove this fact to other parties. In this case, a message authentication code (or some more complicated cryptographic primitive) would have to be used.

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. (E.g., in the case of public-key encryption the receiver publishes a public key whereas in the case of digital signatures this is done by the sender.) Historically, in fact, it has been suggested that digital signatures can be obtained by "reversing" public-key encryption schemes, 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*.

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 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 of the mes-

[1] The 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 encryption meet even minimal notions of security.

sage is again denoted Vrfy. We now formally define the syntax of a digital signature scheme.

DEFINITION 12.1 (signature scheme — syntax): *A* signature scheme *is a tuple of probabilistic polynomial-time algorithms* (Gen, Sign, 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* m *from some underlying message space (that may depend on* pk*). It outputs a signature* σ*, and we write this 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* $b := \mathsf{Vrfy}_{pk}(m, \sigma)$*.*

We require that for every n*, every* (pk, sk) *output by* $\mathsf{Gen}(1^n)$*, and every message* m *in the appropriate underlying plaintext space, it holds that*

$$\mathsf{Vrfy}_{pk}(m, \mathsf{Sign}_{sk}(m)) = 1.$$

We say σ is a *valid signature* on a message m (with respect to some public key pk that is understood form the context) if $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$.

A signature scheme is used in the following way. One party S, who will 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., it can be placed on S's webpage or placed in some public directory. As always in the public-key setting, 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, either intended for one particular party or multiple parties, S computes the signature $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$ and sends (m, σ). Upon receipt of (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 (see Section 4.3) are still possible.

For practical usage of digital signatures, we will want the message space to consist of bit-strings, either of arbitrary length or of length super-polynomial[2]

[2]Signature schemes for *logarithmic*-length messages are relatively easy to construct, and are not that useful for standard applications.

in the security parameter n. Although we will sometimes describe signature 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 $\mathcal{M}$. As far as correctness alone is concerned, this encoding can be arbitrary; for reasons of security it will be necessary for this encoding to be one-to-one, or it should be computationally infeasible to find collisions in the encoding. (See Section 12.4.)

The assumption that parties are able to obtain a legitimate copy of S's public key implies that S is able to send a message (i.e., 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, and 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. (The situation is analogous to the case of private-key encryption, where a secret key must be shared once and this key can then be used to encrypt an unlimited number of messages.) Interestingly, as we will see in Section 12.7, signature schemes themselves are used to ensure the reliable distribution of *other* public keys.

Security of signature schemes. Given a public key pk generated by a signer S, we say 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 the adversary 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 (Definition 4.2), and so we refer the reader there 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 $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n)$**.**

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).
3. *Let* $\mathcal{Q}$ *denote the set of messages whose signatures were requested by* $\mathcal{A}$ *during its execution. The output of the experiment is 1 if* $m \notin \mathcal{Q}$ *and* $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$.

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:

$$\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n)$$

Definition 12.2 is the standard notion of security for digital signature schemes.

12.3 RSA Signatures

In this section we will discuss various signature schemes based on the RSA assumption. We warn the reader that *none of the schemes in this section are known to be secure*; we introduce these schemes mainly to provide some 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. Moreover, a variant of one of the schemes we show here will later be proven secure in a security model that is discussed in detail in Chapter 13.

12.3.1 "Textbook RSA" and its Insecurity

We begin by introducing the "textbook RSA" signature scheme, given this name because (as in the case of textbook RSA encryption) many textbooks still describe RSA signatures this way even though, as we will see, the construction is completely insecure. Let GenRSA be a PPT algorithm that, on input 1^n, outputs (except with negligible probability) a modulus N that is the product of two n-bit primes, along with integers e, d satisfying $ed = 1 \bmod \phi(N)$. The textbook RSA signature scheme is given as Construction 12.3.

It is easy to see that verification of a legitimately-generated signature is always successful since

$$\sigma^e = (m^d)^e = m \bmod N.$$

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 having to obtain any signatures from the legitimate signer. The attack works as follows: given public key $pk = \langle N, e\rangle$, choose arbitrary $\sigma \in \mathbb{Z}_N^*$ and compute $m := \sigma^e \bmod N$; then output the forgery (m, σ). It is immediate that verification succeeds; this is obviously a forgery since no signature on m (in fact, no signatures at all!) were generated by the owner of the public key. We conclude that the textbook RSA signature scheme does not satisfy Definition 12.2.

CONSTRUCTION 12.3

Let GenRSA be as in the text. Define a signature scheme as follows:

- Gen, on input 1^n, runs $\mathsf{GenRSA}(1^n)$ 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 $m \in \mathbb{Z}_N^*$, computes 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 $\sigma \in \mathbb{Z}_N^*$, outputs 1 if and only if
$$m \stackrel{?}{=} [\sigma^e \bmod N].$$

The "textbook RSA" signature scheme.

One may argue that the above does not constitute a realistic attack since the adversary has "no control" over the message m for which it outputs a forgery. Of course, this is irrelevant as far as Definition 12.2 is concerned. Moreover, the adversary does have *some* control over the message m: for example, it can set the last bit of m to ay desired value by carrying out the above attack multiple times, choosing σ at random each time. In any case, 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.

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, 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 and m_2. This constitutes a forgery since m is not equal to m_1 or m_2 (with all but 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. Also, it is dangerous to make assumptions about what messages the signer will or will not be willing to sign. Finally, this attack can be extended to other scenarios: for example, if an adversary is able to obtain valid signatures on some *arbitrary* q messages $M = \{m_1, \ldots, m_q\}$,

then the adversary can output a forgery for any of $2^q - q$ messages obtained by taking products of subsets of M with $|M| \neq 1$.

12.3.2 Hashed RSA

Various modifications of the textbook RSA signature scheme have been proposed in an effort to protect against the attacks described in the previous section. We caution the reader once again that most of these proposals have not been proven secure, and should not be used. Nevertheless, we show one such example here. 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 a model that will be described in detail in Chapter 13.

The basic idea is to take 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 as before except that a description of some function $H : \{0,1\}^* \to \mathbb{Z}_N^*$ is included as part of the public key. A message $m \in \{0,1\}^*$ is now signed by computing

$$\sigma := [H(m)^d \bmod N].$$

(That is, $\hat{m} := H(m)$ is fist computed, followed by $\sigma := [\hat{m}^d \bmod N]$.) Verification of the pair (m, σ) is done 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. Since H must be a collision-resistant hash function, the modified scheme described above is sometimes called the *hashed RSA signature scheme*. The scheme is an example of using the "hash-and-sign" paradigm (introduced in the following section) to the textbook RSA signature scheme.

We stress that 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 on the hashed RSA signature scheme (note that this does *not* mean that these attacks are ruled out):

The no-message attack. The natural way to attempt the no-message attack shown previously is to choose 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.

12.4 The "Hash-and-Sign" Paradigm

In addition to (possibly) offering some protection against certain attacks, the hashed RSA signature scheme has another advantage over 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 some underlying signature scheme) is a standard way to achieve it. We study the security of this approach now.

Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme for messages of length n. (That is, when the security parameter is 1^n, messages of length n can be signed.) Setting the message space to $\{0,1\}^n$ is done for simplicity only, and everything that follows can be modified appropriately for arbitrary message spaces (even message spaces that depend on the public key) as long as the message space is of size super-polynomial in n.[3] Let $\overline{\Pi} = (\overline{\mathsf{Gen}}, \bar{H})$ be a hash function as per Definition 4.9, where the output of H has length n on security parameter 1^n.

CONSTRUCTION 12.4

Let $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ and $(\overline{\mathsf{Gen}}, \bar{H})$ be as in the text.

- Gen', on input 1^n, runs $\mathsf{Gen}(1^n)$ to obtain (pk, sk) and runs $\overline{\mathsf{Gen}}(1^n)$ 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 $sk' = \langle sk, s \rangle$ and a message $m \in \{0,1\}^*$, computes $\sigma' \leftarrow \mathsf{Sign}_{sk}(H^s(m))$.
- Vrfy, on input a public key $pk' = \langle pk, s \rangle$, a message $m \in \{0,1\}^*$, and a signature σ, outputs 1 if and only if $\mathsf{Vrfy}_{pk}(H^s(m), \sigma) \stackrel{?}{=} 1$.

The hash-and-sign paradigm.

We can construct a new signature scheme $\Pi' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Vrfy}')$ for arbitrary-length messages as follows: the public key contains a public key pk output by

[3] Looking ahead, this restriction is needed because collision-resistant hash functions must have super-logarithmic output length.

Gen as well as a key s output by $\overline{\mathsf{Gen}}$; the private key is simply the one corresponding to sk (that was also output by Gen). To sign a message $m \in \{0,1\}^*$, the signer simply computes $\sigma \leftarrow \mathsf{Sign}_{sk}(\bar{H}^s(m))$. Verification is performed by checking that $\mathsf{Vrfy}_{pk}(\bar{H}^s(m), \sigma) \stackrel{?}{=} 1$. 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.

We have the following result:

THEOREM 12.5 *If Π is existentially unforgeable under an adaptive chosen-message attack, and $\overline{\Pi}$ is collision resistant, then Π' is existentially unforgeable under an adaptive chosen-message attack.*

PROOF Let $\mathcal{A}'$ be a probabilistic polynomial-time adversary. In a particular execution of experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n)$, let $pk' = \langle pk, s\rangle$ denote the public key that was 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 $\textsc{coll}_{\mathcal{A}',\Pi'}(n)$ to be the event that in the experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n)$, there exists an $m' \in \mathcal{Q}$ for which $H^s(m') = H^s(m)$.

We have

$$\begin{aligned}
&\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n) = 1] \\
&= \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \textsc{coll}_{\mathcal{A}',\Pi'}(n)] \\
&\quad + \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\textsc{coll}}_{\mathcal{A},\Pi'}(n)] \\
&\leq \Pr[\textsc{coll}_{\mathcal{A}',\Pi'}(n)] + \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\textsc{coll}}_{\mathcal{A},\Pi'}(n)]. \quad (12.1)
\end{aligned}$$

We will show that both terms in Equation (12.1) are negligible, which will complete the proof. Intuitively, the first term is negligible by collision resistance of $(\overline{\mathsf{Gen}}, \bar{H})$, and the second term is negligible by security of Π.

Consider the following PPT algorithm $\mathcal{C}$ who attempts to find a collision in scheme $\overline{\Pi}$:

> **Algorithm $\mathcal{C}$:**
> The algorithm is given s as input (the security parameter n is implicit).
>
> - Compute $\mathsf{Gen}(1^n)$ 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 $\overline{\mathsf{Gen}}(1^n)$ to obtain s, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{C}$ is distributed exactly as its view in experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n)$. Since $\mathcal{C}$ outputs a collision exactly when $\textsc{coll}_{\mathcal{A}',\Pi'}(n)$ occurs, we have

$$\Pr[\mathsf{Hash\text{-}coll}_{\mathcal{C},\overline{\Pi}}(n) = 1] = \Pr[\textsc{coll}_{\mathcal{A}',\Pi'}(n)].$$

Because $\overline{\Pi}$ is collision resistant, we conclude that $\Pr[\textsc{coll}_{\mathcal{A}',\Pi'}(n)]$ must be negligible.

Consider next 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 $\mathsf{Sign}_{sk}(\cdot)$.
>
> - $\mathcal{A}$ computes $\overline{\mathsf{Gen}}(1^n)$ to obtain s, and sets $pk' = \langle pk, s\rangle$.
> - $\mathcal{A}$ runs $\mathcal{A}'$ on input pk'. When $\mathcal{A}'$ requests the ith signature on a message $m_i \in \{0,1\}^*$, this is answered as follows: (1) $\mathcal{A}$ computes $\hat{m}_i := H^s(m_i)$; then (2) $\mathcal{A}$ obtains a signature σ_i on $\hat{m}_i$ from its own signing oracle, and gives σ_i to $\mathcal{A}'$.
> - Eventually, $\mathcal{A}'$ outputs (m, σ). Adversary $\mathcal{A}$ outputs $(H^s(m), \sigma)$.

Consider experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}\mathcal{A}, \Pi(n)$. In this experiment, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{A}$ is distributed exactly as its view in experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n)$. Furthermore, it can be easily verified that whenever both $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n) = 1$ and $\overline{\textsc{coll}}_{\mathcal{A},\Pi'}(n)$ occur, $\mathcal{A}$ outputs a forgery. (That σ is a valid signature on $H^s(m)$ with respect to pk is immediate. The fact that $\overline{\textsc{coll}}_{\mathcal{A},\Pi'}(n)$ occurs means that $\hat{m} = H^s(m)$ was never asked by $\mathcal{A}$ to its own signing oracle.) Therefore,

$$\Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n) = 1] \geq \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}',\Pi'}(n) \wedge \overline{\textsc{coll}}_{\mathcal{A},\Pi'}(n)],$$

and so the latter probability must be negligible by security of Π. This concludes the proof of the theorem. ∎

An analogue of Theorem 12.5 holds for the case of message authentication codes, and gives an alternate way of constructing variable-length MACs from fixed-length ones (albeit under the additional assumption that collision-resistant hash functions exist, which is not needed for Theorem 4.6).

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 definitions of security.

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 from any one-way function. (In this section, we rely only on the definition of one-way functions given in Section 7.4.1, and do not need any of the material developed in Chapter 6.) We will use one-time signature schemes in the following section to construct signature schemes satisfying Definition 12.2.

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* $\mathsf{Sign}_{sk}(\cdot)$. *(That is,* $\mathcal{A}$ *requests a signature on a* single *message* m'*.)* $\mathcal{A}$ *then outputs* (m, σ) *where* $m \neq m'$.
3. *The output of the experiment is 1 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 simply 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 quite simple, and we illustrate it for the case of signing a 3-bit message. Let f be a one-way function.

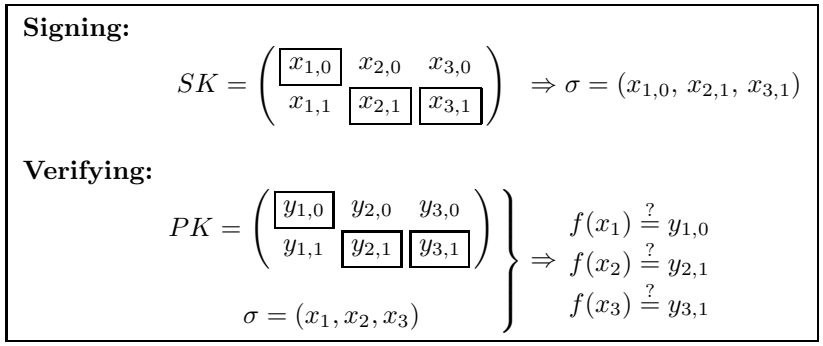

FIGURE 12.1: The Lamport scheme used to sign the message $m = 011$.

(Recall this means, informally, that f is easy to compute but hard to invert; see Definition 7.66.) The public key will consist of $2 \cdot 3 = 6$ elements $y_{1,0}$, $y_{1,1}$, $y_{2,0}$, $y_{2,1}$, $y_{3,0}$, $y_{3,1}$ in the range of f; the private key will contain the corresponding pre-images $x_{1,0}$, $x_{1,1}$, $x_{2,0}$, $x_{2,1}$, $x_{3,0}$, $x_{3,1}$ under f. 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 done 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, and formally as Construction 12.7.

CONSTRUCTION 12.7

- Gen, on input 1^n, proceeds 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 $m = m_1 \cdots m_\ell$, outputs 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 $\sigma = (x_1, \ldots, x_\ell)$, outputs 1 if and only if $f(x_i) = y_{i,m_i}$ for $1 \le i \le \ell$.

Lamport's signature scheme for messages of length $\ell = \ell(n)$.

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.*

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. Might it become easier to compute a signature on some message m after being given a signature on a different message m'? Note that 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 easier even when given $\{x_{i,b}\}$ for all $(i,b) \neq (i^*,b^*)$ (since $\{x_{i,b}\}_{(i,b)\neq(i^*,b^*)}$ 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 $\mathcal{A}$ *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 i^{th} 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 as input (with n implicit).

1. Choose random $i^* \leftarrow \{1,\ldots,\ell\}$ and $b^* \leftarrow \{0,1\}$. Set $y_{i^*,b^*} := y$.
2. For $i \in \{1,\ldots,\ell\}$ 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^*}.

Note that 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 random $x \leftarrow \{0,1\}^n$ and setting $y := f(x)$ (cf. Definition 7.66). Imagine for a moment 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 $\mathcal{A}$ output a forgery at all, 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 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 $\varepsilon(n)/2\ell(n)$. That is,

$$\Pr[\mathsf{Invert}_{\mathcal{I},f}(n) = 1] \geq \varepsilon(n)/2\ell(n).$$

Because f is a one-way function, there is a negligible function negl such that

$$\mathsf{negl}(n) \geq \Pr[\mathsf{Invert}_{\mathcal{I},f}(n) = 1] \geq \varepsilon(n)/2\ell(n).$$

Since ℓ is polynomial, ε must be negligible. This completes the proof. ■

COROLLARY 12.9 *If one-way functions exist, then there exist one-time signature schemes.*

We remark that the Lamport scheme is completely insecure if used to sign more than one message; see Exercise 12.2.

12.6 * Signatures from Collision-Resistant Hashing

We have not yet seen any signature schemes that are 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 the cryptographic assumptions we have introduced thus far. The construction relies only 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. It remains open to develop other efficient signature schemes that can be proven secure in the "standard" model we have been using until now.

We build up to our final construction in stages. In Section 12.6.1 we define the notion of a *stateful* signature scheme, 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* $|pk| \geq n$*, and that* n *can be determined from* pk*.*

2. *The* signing algorithm Sign *takes as input a private key* sk*, a value* s_{i-1}*, and a message* m*. It outputs a signature* σ *along with a value* s_i*, and we write this as* $(\sigma, s_i) \leftarrow \mathsf{Sign}_{sk, s_{i-1}}(m)$*.*

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*, and we write this as* $b := \mathsf{Vrfy}_{pk}(m, \sigma)$*.*

We require that for every n*, every* (pk, sk, s_0) *output by* $\mathsf{Gen}(1^n)$*, and any sequence of messages* $m_1, \ldots, m_\ell$*, if we compute* $(\sigma_i, s_i) \leftarrow \mathsf{Sign}_{sk, s_{i-1}}(m_i)$ *recursively for* $i \in \{0, \ldots, \ell\}$ *then*

$$\mathsf{Vrfy}_{pk}(m_\ell, \sigma_\ell) = 1.$$

We emphasize that the state is not needed to verify a signature. Signature schemes which do not maintain state are sometimes called *stateless* to distinguish them from stateful schemes.

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 be used to 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 will only be informal.) Simply let 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 initial state is set to 1.

To sign a message m using the secret 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 out at 1, this means that the i^{th} 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 public-/private-key pair 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 to be signed *to be fixed in advance*, at the time of key generation. This is a potentially severe limitation in practice, as once this 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. The scheme is also not very efficient, as the public and private keys have length 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.

In the chain-based scheme, the public key will consist of just a single public key pk_1 generated using Gen, and the private key will contain 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 stores $(m_1, pk_2, sk_2, \sigma_1)$ as part of its state.

In general, when it comes time to sign the i^{th} 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 i^{th} 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 includes $(m_i, pk_{i+1}, sk_{i+1}, \sigma_i)$ as part of its state. See Figure **??** 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 a 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_1. 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 $j \in \{1, \ldots, i\}$. (It may help to refer to Figure **??**.)

It is not hard to be convinced — at least on an intuitive level — that the 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 secure a more efficient scheme in the next section, we do not give a formal proof of security for the chain-based scheme here.

An important point regarding the chain-based scheme that we have neglected to mention until now is that each public key pk_i in this scheme 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. If we apply the "hash-and-sign" paradigm from Section 12.4 to the Lamport scheme, however, 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 exist one-time signature schemes that can sign 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 12.5). ■

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. Primary among these is that there does not appear to be any easy way to eliminate the state in this scheme (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) of degree 1 and depth equal to the number of messages signed thus far (cf. Figure **??**). 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 tree of depth n having 2^n leaves. As before, the signer will add nodes to the tree "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 labeled with the binary string $w \in \{0,1\}^{<n}$ 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.

Associated with each node w will be a public key pk_w generated using 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:

1. Generates (as needed) public keys for all nodes on the path from the root to the leaf labeled m. (Note that 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. "Certifies" the path from the root to the leaf labeled m by computing a signature on $pk_{w0} \| pk_{w1}$ with respect to pk_w, for each string w that is a proper prefix of m.
3. "Certifies" m itself by computing a signature on m with respect to pk_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 **??**. Additionally, the signer updates its state by storing any public-/private-key pairs generated as part of the above signing process. A formal description of this scheme is given as Construction 12.12.

Notice that each of the underlying keys in this scheme is being used to sign a *pair* of public keys (except at a leaf). Thus, we need out one-time

CONSTRUCTION 12.12

Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme. For m a binary string, 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 scheme $\Pi^* = (\mathsf{Gen}^*, \mathsf{Sign}^*, \mathsf{Vrfy}^*)$ as follows:

- Gen^*, on input 1^n, computes $(pk_\varepsilon, sk_\varepsilon) \leftarrow \mathsf{Gen}(1^n)$ and outputs the public key pk_ε. The secret key and initial state are sk_ε.
- Sign^*, on input a message $m \in \{0,1\}^n$, does the following.
  1. For $i = 0$ to $n-1$:
     - If the values $pk_{m|_i 0}, pk_{m|_i 1}$, and $\sigma_{m|_i}$ are not yet included as part of the state, then compute the values $(pk_{m|_i 0}, sk_{m|_i 0}) \leftarrow \mathsf{Gen}(1^n)$, $(pk_{m|_i 1}, sk_{m|_i 1}) \leftarrow \mathsf{Gen}(1^n)$, and $\sigma_{m|_i} \leftarrow \mathsf{Sign}_{sk_{m|_i}}(pk_{m|_i 0} \,\|\, pk_{m|_i 1})$, and store all these values as part of the state.
  2. If σ_m is not 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 signature $\left(\left\{\sigma_{m|_i}, pk_{m|_i 0}, pk_{m|_i 1}\right\}_{i=0}^{n-1},\ \sigma_m\right)$, message m, and public key pk_ε, outputs 1 if and only if both:
  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.

signature scheme Π to be capable of signing messages longer than the public key. Fortunately, we know from Lemma 12.11 that such schemes can be constructed based on collision-resistant hash functions.

Before proving security of this tree-based approach, we note that it improves on the chain-based scheme in a number of respects. It still allows for signing an unbounded number of messages.[4] 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 that can sign messages of arbitrary length. Then Π^* as in Construction 12.12 is existentially unforgeable under an adaptive chosen-message attack.*

[4] 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.

PROOF 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 $\ell(n) \stackrel{\text{def}}{=} 2n\ell^*(n) + 1$. Define

$$\lambda(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}^*,\Pi^*}(n) = 1].$$

Consider the following PPT adversary $\mathcal{A}$ attacking signature scheme Π:

Adversary $\mathcal{A}$:
$\mathcal{A}$ is given as input a public key pk (the security parameter n is implicit).

- Choose 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}. Then compute[5] a signature $\sigma_{m|_i}$ on $pk_{m|_i 0} \,\|\, pk_{m|_i 1}$ with respect to $pk_{m|_i}$.
  2. If σ_m is not yet defined, compute a signature σ_m on m with respect to pk_m (see footnote 5).
  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 $\left(\left\{\sigma'_{m|_i}, pk'_{m|_i 0}, pk'_{m|_i 1}\right\}_{i=0}^{n-1},\ \sigma'_m\right)$. 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 $pk^i = pk_{m|_j} = pk'_{m|_j}$. 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).

[5] 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.

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 $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A}^*,\Pi^*}(n)$.[6] 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:

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.

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.

We thus 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, but to a much more limited extent than the chain-based scheme of the previous section, in a sense we now make precise. In scheme Π^*, we could 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

[6] One point that is not completely trivial to check is that $\mathcal{A}$ 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.

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 requires *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 $(pk_w, sk_w) := \mathsf{Gen}(1^n; r_w)$. Similarly, the signer can store r'_w and 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 (say) can now be generated by the following two-step process:

1. Compute $r_w := F_k(w)$.[7]
2. Compute $(pk_w, sk_w) := \mathsf{Gen}(1^n; r_w)$ (as before).

(Key k' is used to generate the $\{r'_w\}$.) This gives a *stateless* signature scheme in which key generation (as well as signing and verifying) can be done in polynomial time; 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. 12.5), and the latter implies the existence of pseudorandom functions (see Chapter 6), we have:

THEOREM 12.14 *If collision-resistant hash functions exist, there exists a (stateless) signature scheme that is existentially unforgeable under an adaptive chosen-message attack.*

In fact, it is known that signature schemes satisfying Definition 12.2 can be constructed based on the (minimal) assumption that one-way functions exist; a proof of this result is beyond the scope of this book.

12.7 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

[7] We assume 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.

brings us full-circle in our discussion of public-key cryptography: in this and the previous three chapters we have seem 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 since 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 "boot-strap" the secure distribution of arbitrarily-many other public keys. Thus, the problem of secure key public key distribution need only be solved *once* (theoretically speaking, at least).

The key idea is the notion of a *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. We remark that in practice a certificate should unambiguously identify the party holding a particular public key and so a more descriptive term than "Bob" would be used, for example, Bob's 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 to accept pk_B as Bob's public key even assuming the certificate is valid. Fully specifying such details (and others) gives a *public-key infrastructure* (PKI) that enables 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 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 done 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 very difficult step need only be 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 "hard-wired" into the web browser's code, and the web browser can be programmed to automatically verify certificates (with respect to the hard-coded public key) as they arrive. (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 means for some party Bob to obtain 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 will 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 a valid one 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. Note that this needn't even 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 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 down then no new certificates can be issues, and the load on the CA may be very high if many parties want to obtain certificates at once.

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 it trusts. There is no harm in having Bob obtain a certificate from every CA (there this may result in some inconvenience or 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 "hard-wired" in, and the default setting is for all 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 reputation, well-established companies along with other, new 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. Now Bob, in turn, can issue his own certificates for other parties. For example, Bob may issue a certificate for Alice of the form

$$\mathsf{cert}_{B \to A} \stackrel{\text{def}}{=} \mathsf{Sign}_{sk_B}(\texttt{`Alice's key is } pk_A\texttt{'}).$$

Now, say 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 names 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 for trustworthy people, then Dave may accept pk_A as being the authentic key of Alice.

Note that 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*. We remark that 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 will issue 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 for 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 must 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 some minimal 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 a bunch of PGP users get together (at a conference, say) and give each other authentic copies of their public keys and and issue certificates for each other. Note that in general the users at a key-signing party may not know each other, but then 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 really 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 a bunch of 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 on at all is the fact that certificates should generally not be valid indefinitely. An employee may leave a company, no longer be 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; to verify, 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 their 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 12.8.) 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 *revocation list* containing the serial numbers of all revoked certificates, and sign this entire list along with the current date. The singed 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 lag time 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 [50], Rabin [106, 107], Rivest, Shamir, and Adleman [110], and Goldwasser, Micali, and Yao [72]. Lamport's one-time signature scheme was published in 1979 [88], though it was already described in [50].

Goldwasser, Micali, and Rivest [71] 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 would be *impossible* to achieve.) Goldreich [64] 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 that we have not introduced in this book. The interested reader can consult [62, 41, 56].

A tree-based construction similar in spirit to Construction 12.12 was suggested by Merkle [94, 95], though a tree-based approach was also used in [71]. Naor and Yung [98] showed that one-way permutations suffice for constructing one-time signatures that can sign messages of arbitrary length, and this was improved by Rompel [111, 82] who showed that one-way functions are sufficient. As we have seen in Section 12.6.2, one-time signatures of this sort cam be used to construct signature schemes that are existentially unforgeable under an adaptive chosen-message attack.

Goldreich [66, Chapter 6] and Katz [81] provide a more extensive treatment of signature schemes than what is covered here.

The notion of certificates was first described by Kohnfelder [85] in his undergraduate thesis. Public-key infrastructures are discussed in greater detail in [84, Chapter 15] and [10]. Ellison and Schneier [53] discuss some reasons why PKI is not a panacea for identity management.

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 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.

12.3 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 $m\{0,1\}^{\ell'}$ 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?

12.4 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) Assuming the existence of collision-resistant hash functions, construct a strong one-time signature scheme.

Hint: Use a particular one-way function in Lamport's scheme.

12.5 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 computing an inverse of $y = H^s(x)$ does not guarantee a collision. But it does yield a collision most of the time...

12.6 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.

12.7 Prove Theorem 12.14.

12.8 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.

12.9 Consider the following "improvement" for handling revocation: instead of signing a revocation list containing the serial numbers of all revoked certificates at the end of each day, simply sign the serial number of a revoked certificate immediately and immediately post this signed serial number on a public webpage (adding it to the end of the list of previously-signed serial numbers).

Explain why this approach is insecure, in that an adversary who steals a user's private key can prevent the relevant certificate from ever being recognized as invalid by another user. Assume there are no authenticated channels in the system.

Chapter 13

Public-Key Cryptosystems in the Random Oracle Model

In the previous two chapters, we have seen constructions of digital signatures and public-key encryption schemes based on a variety of assumptions. For the most part, however, the provably-secure schemes we have discussed and analyzed are not particularly efficient. More to the point:

- In Section 10.4.3 we have seen an encryption scheme that can be proven secure based on the RSA assumption (cf. Theorem 10.17), 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.
- We have not shown any public-key encryption schemes secure against chosen-ciphertext attacks. Though efficient schemes based on certain assumptions are known (these are, however, beyond the scope of this book), there is no known scheme based on RSA that is even remotely practical.
- We saw only a single example of a digital signature scheme satisfying the desired level of security given in Definition 12.2. This construction, shown in Section 12.6.2, is not very practical. In fact, no secure signature scheme based on RSA with efficiency comparable to the textbook RSA signature scheme is known.

We stress that the above statements remain true even given an efficient pseudorandom function (that could be instantiated in practice using a block cipher such as DES or AES) and/or an efficient collision-resistant hash function (that could be instantiated using a cryptographic hash function such as SHA-1).

We conclude that for most "standard" cryptographic assumptions (such as the RSA, factoring, or DDH assumptions), there are few or no public-key cryptosystems that are both (1) efficient enough to be used in practice, yet (2) can be proven secure based on these assumptions. This state of affairs presents a challenge to cryptographers, who continue to work at improving the efficiency of existing solutions, proposing new assumptions, and showing limitations to the best possible efficiency that can be achieved. In the meanwhile, we are left in practice with the question of what schemes to use. While one might argue

that we should simply choose the "best" schemes that are currently available (according to whatever criteria one likes), the reality appears to be that people for the most part would rather use *nothing* than use an inefficient scheme. (Furthermore, in some cases existing solutions are not remotely practical even if one is willing to sacrifice some efficiency.) Something must give.

One possibility, of course, is simply 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 seems (to us) "hard to break" we potentially 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

Another 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.'

It should be stressed that no one seriously claims that any such oracle exists (although there have been suggestions that a random oracle could be implemented in practice by a trusted party). Rather, the random oracle model provides a formal *methodology* 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 that the world contains a random oracle, and construct and analyze a 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 or MD5, 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 simulating 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 simulating a random oracle, nor is it clear that, as stated, 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.

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. Note that no one "knows" 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.

Security guarantees in the random oracle model are a bit different from the security guarantees we are familiar with from the rest of the book, as we

now discuss. Fix some cryptographic scheme Π that relies on an oracle; that is, the scheme requires the honest parties running Π to query the oracle at various points during their execution. A general definition of (computational) security for Π has the following form:

> *For any polynomial-time adversary $\mathcal{A}$, the probability that some event occurs when $\mathcal{A}$ interacts with Π is negligible.*

The difference is that in all the definitions we have seen thus far, the probability in question is taken over *the random choices made by the honest parties running* Π. (If $\mathcal{A}$ is randomized then the probability is taken over the random choices of $\mathcal{A}$ as well, but this is less important for the current discussion.) In the random oracle model, the probability in question is taken over these random choices *as well as the random choice of a function H for the oracle*. (This should become more clear from the examples given in the following section.) It is for this reason that we speak of H as a *random* oracle.

A brief digression is in order regarding what it means to choose a function (uniformly) at random. Any function H mapping n_1-bit inputs to n_2-bit outputs can be viewed as a table indicating for each possible input $x \in \{0,1\}^{n_1}$ the corresponding output value $H(x) \in \{0,1\}^{n_2}$. Using lexicographic order for the inputs, this means that any such function can be represented by a string of length $2^{n_1} \cdot n_2$ bits, and conversely that every string of this length can be viewed as a function mapping n_1-bit inputs to n_2-bit outputs. An immediate corollary is that there are exactly $U \stackrel{\text{def}}{=} 2^{n_2 \cdot 2^{n_1}}$ 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 $y \in \{0,1\}^{n_2}$ 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 identical.

Returning to our discussion of security in the random oracle model, note that even if some scheme Π is proven secure in this model, security is not

guaranteed for any *particular* function H but is instead only guaranteed "on the average" over random choice of H. (This is exactly 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 on average over these random choices.) This indicates one reason why it is difficult to argue that any concrete instantiation of the oracle H yields a real-world implementation of Π that is actually secure.

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 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 computed.
3. $\mathcal{A}$ is given y, and succeeds if it outputs a value x' such that $H(x') = y$.

We now argue why this is true. Assume without loss of generality that the value x' that $\mathcal{A}$ outputs was previously queried by $\mathcal{A}$ to the oracle. Assume further than 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 once it knows 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 ℓ be the number of queries $\mathcal{A}$ makes to the oracle, with $\ell = \mathsf{poly}(n)$ 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 $\ell/2^{n_1}$. 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 $\ell/2^{n_2}$. Since $n_1, n_2 \geq n$ the probability that $\mathcal{A}$ succeeds in the latter game is at most $2\ell/2^n = \mathsf{poly}(n)/2^n$, 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 oracles queries of $\mathcal{A}$ be $x_1, \ldots, x_\ell$, with $\ell = \mathsf{poly}(n)$, 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 done "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 Section A.4 we see that $\mathcal{A}$ succeeds with probability $\mathcal{O}(\ell^2/2^{n_2})$, 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 $b = b'$.

We stress that $\mathcal{A}$ can access H in addition to the function oracle provided to it by the experiment in step 2. In Exercise 13.1 you are asked to show that the construction above indeed gives a pseudorandom function.

It is worth reflecting that the use of an "oracle" and a "random function" in step 2 (and, indeed, back in Chapter 3 when we first defined pseudorandom functions) is fundamentally *different* from the use of a random oracle/function in the random oracle model. In the context of pseudorandom functions, the oracle and random function are used *as a definitional tool* but need not exist for the primitive itself to be realized. In the random oracle model, in contrast, the random oracle is *used as part of the construction* and so something must take the place of the oracle for the construction to be realized.

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 only making 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 oracle and hence no way to define what it means to "query an oracle/evaluate a hash function" polynomially-many times.

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. We sketch them 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 it is actually quite different. If G is a pseudorandom generator then $G(x)$ is pseudorandom *assuming x is chosen uniformly at random and is unknown to the observer*. If x is known then trivially $G(x)$ is too. For H a random oracle, however, $H(x)$ is (truly) random as long as the adversary has not queries x. This is true even if x is known. Furthermore, if x is not chosen uniformly at random, but is chosen with enough entropy to make guessing x difficult, 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 often 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.[1] As part of the reduction, the random oracle that $\mathcal{A}$ interacts with must be simulated as part of the reduction. That is: $\mathcal{A}$ will submit queries to and receive answers from what it believes to be the oracle, but what is actually the reduction itself. This turns out to give a lot of power. For starters:

> *As part of the reduction, we may choose values for the output of H "on-the-fly" (as long as these values are correctly distributed, i.e., uniformly random).*

This is sometimes called "programmability". Although this may not seem like programmability confers any advantage, it does, as perhaps illustrated best by the proof of Theorem 13.11 will illustrate. Another advantage deriving from the fact that the reduction gets to simulate the random oracle is

> *The reduction gets to "see" the queries that $\mathcal{A}$ makes to the random oracle.*

(Note that this does contradict the fact, mentioned earlier, that queries to the random oracle are supposed to be "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.

13.1.2 Is the Random Oracle Methodology Sound?

With the mechanics of the random oracle model behind us, we can turn to more fundamental questions such as: *What do proofs of security in the random oracle model guarantee in the real world?*, and: *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 all sides of this discussion.

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

[1] In contrast, the proofs in the previous section were information-theoretic and did not use reductions.

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 $H(x)$ is supposed to be "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 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.)

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 any given scheme Π one could, after analyzing Π, decide to assume that Π is secure when instantiated with SHA-1, it is much less reasonable to assume that SHA-1 can take the place of the random oracle in *any* 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.)

It is worth emphasizing 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 should mean (while we do have such definitions for the latter two statements). In particular, a random oracle is not the same thing 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, and 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.

In support of the random oracle model. Given all the problems with the random oracle model, why do we cover the random oracle model at all? More to the point: why has the random oracle been so influential in the development of modern 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 have 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 for the above 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 might disagree with the above, we offer the following in support of this conviction:

- A proof of security for a given scheme in the random oracle model indicates that the 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 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 itself. 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 no real-world attacks on any "natural" schemes proven secure in the random oracle model* (we do not include here attacks on "contrived" schemes like that of Exercise 13.2). 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. Understanding exactly what such proofs guarantee in the real world remains, in our minds, one of the most important research questions facing cryptographers today.

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, things are complicated by a number of issues such as (to name two)

- Cryptographic hash functions almost all use a Merkle-Damgård construction (cf. Section 4.6.4), and can therefore be distinguished relatively easy from a random oracle taking variable-length inputs. (In contrast, there are no known attacks on such hash function when restricting to fixed-length inputs.)
- Frequently, it is necessary for the output of a 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. (When we need such an oracle for our proofs, we will simply assume that one exists.)

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 can arise.

13.2 Public-Key Encryption in the Random Oracle Model

We show in this section 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 10.17) was both inefficient and difficult to prove secure (indeed, we offered no proof). In the random oracle model, things become significantly easier. As usual, GenRSA denotes 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 $ed = 1 \bmod \phi(N)$.

CONSTRUCTION 13.1

Let GenRSA be as in the text, and let $H : \{0,1\}^{2n} \to \{0,1\}^{\ell(n)}$ be a function for ℓ an arbitrary polynomial.

Key generation. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d), where N is of length $2n$ and elements of $\mathbb{Z}_N^*$ are represented by $2n$-bit strings. The public key is $\langle N, e\rangle$ and the private key is (N, d).

Encryption. To encrypt a message $m \in \{0,1\}^{\ell(n)}$ with respect to the public key $\langle N, e\rangle$, choose random $r \leftarrow \mathbb{Z}_N^*$ and output the ciphertext

$$\langle [r^e \bmod N],\ H(r) \oplus m \rangle .$$

Decryption. To decrypt ciphertext $\langle c_1, c_2\rangle$ using private key $\langle N, d\rangle$, 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.

We can argue intuitively that the scheme is CPA-secure in the random oracle model (under the RSA assumption) 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 point of view of the adversary. 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 says that 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 chosen-plaintext attacks.*

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 $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$:

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. We also use Succ as shorthand for the event that $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$. Then

$$\begin{aligned}\Pr[\mathsf{Succ}] &= \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Succ} \wedge \mathsf{Query}] \\ &\leq \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Query}]\end{aligned}$$

where all probabilities are taken over the randomness used in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. We now show that $\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2}$ and that $\Pr[\mathsf{Query}]$ is negligible. The theorem follows.

CLAIM 13.3 *If H is modeled as a random oracle,* $\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2}$.

If $\Pr[\overline{\mathsf{Query}}] = 0$ then the claim is immediate. Otherwise, we have

$$\begin{aligned}\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] &= \Pr\left[\mathsf{Succ} \mid \overline{\mathsf{Query}}\right] \cdot \Pr[\overline{\mathsf{Query}}] \\ &\leq \Pr\left[\mathsf{Succ} \mid \overline{\mathsf{Query}}\right].\end{aligned}$$

Furthermore, $\Pr\left[\mathsf{Succ} \mid \overline{\mathsf{Query}}\right] = \frac{1}{2}$. 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 chosen at random. 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[Query], which must be negligible because the RSA problem is hard relative to GenRSA.

Formally, consider the following algorithm $\mathcal{A}'$:

Algorithm $\mathcal{A}'(N, e, c_1)$

1. Choose random $k^* \leftarrow \{0,1\}^{\ell(n)}$. (We imagine that $\mathcal{A}'$ implicitly sets $H(r) = k^*$, where $r \stackrel{\text{def}}{=} [c_1^{1/e} \bmod N]$. Note, however, that $\mathcal{A}'$ does not know 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 random oracle query $H(x)$, answer it as follows:
   - If there is an entry (x, k) in the table, return k.
   - If $x^e = c_1 \bmod N$, return k^* and store (x, k^*) in the table. (Note that in this case we have $x = r$, for 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 random $b \leftarrow \{0,1\}$ and set $c_2 := k^* \oplus m_b$. Give $\mathcal{A}$ the ciphertext $\langle c_1, c_2\rangle$.
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 $x_i^e = c_1 \bmod N$, 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 $c_1 \leftarrow \mathbb{Z}_N^*$ 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; 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. ■

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, e.g., the RSA or factoring assumptions. (There are, however, practical CCA-secure public-key encryption schemes based on the DDH assumption.) 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 $\Pi' = (\mathsf{Enc}', \mathsf{Dec}')$ be a *private-key* encryption scheme for messages of length $\ell(n)$ whose keys are (without loss of generality) of length n. Looking ahead, we will want Π' to be secure against chosen-ciphertext attacks; as shown in Section 3.7, efficient private-key encryption schemes satisfying this notion can be constructed relatively easily.

CONSTRUCTION 13.5

Let Π' be a private-key encryption scheme as described in the text, and GenRSA be as in the previous section. $H : \{0,1\}^{2n} \to \{0,1\}^n$ is a function.

Key generation. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.

Encryption. To encrypt a message $m \in \{0,1\}^{\ell(n)}$ with respect to the public key $\langle N, e\rangle$, first choose random $r \leftarrow \mathbb{Z}_N^*$ and compute $k := H(r)$. Output the ciphertext

$$\left\langle [r^e \bmod N],\ \mathsf{Enc}'_k(m)\right\rangle.$$

Decryption. To decrypt ciphertext $\langle c_1, c_2\rangle$ using private key $\langle N, d\rangle$, first compute $r := [c_1^d \bmod N]$ and set $k := H(r)$. Output $\mathsf{Dec}'_k(c_2)$.

CCA-secure RSA encryption in the random oracle model.

It is instructive to compare the above with Construction 13.1. In both cases, the sender chooses a random $r \leftarrow \mathbb{Z}_N^*$ and sends $c_1 = [r^e \bmod N]$ as part of the ciphertext. The difference between the two schemes, on a conceptual level, is that in Construction 13.1 the sender encrypts the message m using the key $H(r)$ and a private-key encryption scheme that has *indistinguishable encryptions in the presence of an eavesdropper*. (Construction 13.1 utilizes the

one-time pad encryption scheme since, letting $k := H(r)$ there, the second component of the ciphertext is computed as $k \oplus m$. We remark that an analogous proof can be given when Construction 13.1 is instantiated with an arbitrary CPA-secure private-key encryption scheme; see Exercise 13.3.) Here, in contrast, the sender encrypts the message m using the key $k = H(r)$ and a private-key encryption scheme that has *indistinguishable encryptions under a chosen-ciphertext attack*.

The intuition for the proof of security, when H is modeled as a random oracle, is roughly as in the previous section. In the proof, we will again distinguish between the case when the adversary does not query r to the random oracle H and the case when it does. In the first case, the adversary learns nothing about the key $k = H(r)$ and so we have reduced the security of our 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 fact is a bit more complex than in the previous section because we must now show how it is possible to simulate decryption oracle queries without knowing the private key. We show how to do this by "programming" the random oracle in an appropriate way.

THEOREM 13.6 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, Construction 13.5 has 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 the reader's 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 $H(\cdot)$ and the decryption oracle $\mathsf{Dec}_{\langle N,d\rangle}(\cdot)$. Eventually $\mathcal{A}$ outputs two messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$,*
3. *Random $b \leftarrow \{0,1\}$ and $r \leftarrow \mathbb{Z}^*_N$ are chosen, and $\mathcal{A}$ is given the ciphertext $\langle r^e \bmod N, \mathsf{Enc}'_{H(r)}(m_b)\rangle$. Adversary $\mathcal{A}$ may continue to query $H(\cdot)$ and the decryption oracle, though it may not query the latter on the ciphertext it was given.*
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 $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$, let Query denote the event that, at any point during its execution, $\mathcal{A}$ queries r to the random oracle H. We

also use Succ as shorthand for the event that $b' = b$. Then

$$\begin{aligned}\Pr[\mathsf{Succ}] &= \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Succ} \wedge \mathsf{Query}] \\ &\leq \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Query}],\end{aligned}$$

where all probabilities are taken over the randomness used in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. We now show that there exists a negligible function negl such that

$$\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2} + \mathsf{negl}(n),$$

and that $\Pr[\mathsf{Query}]$ is negligible. The theorem follows.

CLAIM 13.7 *If 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*

$$\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2} + \mathsf{negl}(n).$$

The proof now is much more involved than the proof of the corresponding claim in the previous section. This is because, as discussed in the intuition preceding this theorem, Construction 13.1 uses the (perfectly-secret) one-time pad encryption scheme 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 3.31):

Adversary $\mathcal{A}'(1^n)$

$\mathcal{A}'$ *has access to an encryption oracle* $\mathsf{Enc}'_k(\cdot)$ *and a decryption oracle* $\mathsf{Dec}'_k(\cdot)$ *for some key* k

1. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). Choose $r \leftarrow \mathbb{Z}^*_N$ and set $c_1 := [r^e \bmod N]$.
   /* *We will imagine that $\mathcal{A}'$ implicitly sets $H(r) = k$, though $\mathcal{A}'$ does not know 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 $\langle \bar{c}_1, \bar{c}_2\rangle$ to its decryption oracle, answer it as follows:
   - If $\bar{c}_1 = c_1$, then $\mathcal{A}'$ queries $\bar{c}_2$ to its own decryption oracle and returns the result to $\mathcal{A}$.
   - If $\bar{c}_1 \neq c_1$, compute $\bar{r} := [\bar{c}_1^d \bmod N]$. Then compute $\bar{k} := H(\bar{r})$ using the procedure discussed below. Return the result $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$ to $\mathcal{A}$.

   When the value $H(\bar{r})$ 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 $H(\bar{r})$ as follows:

- If there is an entry $(\bar{r}, \bar{k})$ in the table, return $\bar{k}$.
- Otherwise, choose a random $\bar{k} \leftarrow \{0,1\}^n$ return it, and store $(\bar{r}, \bar{k})$ in the table.

3. At some point, $\mathcal{A}$ outputs $m_0, m_1 \in \{0,1\}^{\ell(n)}$. Adversary $\mathcal{A}'$ outputs these same messages, and is given in return a ciphertext c_2. Then $\mathcal{A}'$ gives the ciphertext $\langle c_1, c_2 \rangle$ to $\mathcal{A}$, and continues 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 the ciphertext c_2 to its own decryption oracle after it is given this ciphertext in step 3; the only way this could happen would be if $\mathcal{A}$ submitted $\langle c_1, c_2 \rangle$ to *its* decryption oracle, but this is not allowed.

Let $\Pr'[\cdot]$ refer to the probability of an event in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$. Define Succ and Query as above; that is, Succ is the event that $b' = b$, and Query is the event that $\mathcal{A}$ queries 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. So

$$\Pr'[\mathsf{Succ}] \geq \Pr'[\mathsf{Succ} \wedge \mathsf{Query}] = \Pr[\mathsf{Succ} \wedge \mathsf{Query}].$$

(The inequality is trivial, and the equality follows from the observation we just made.) Because Π' is CCA-secure, there exists a negligible function negl such that

$$\frac{1}{2} + \mathsf{negl}(n) \geq \Pr'[\mathsf{Succ}] \geq \Pr[\mathsf{Succ} \wedge \mathsf{Query}],$$

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. In the formal proof, however, additional difficulties arise due to the fact that the decryption queries of $\mathcal{A}$ must somehow be answered *without knowledge of the private (decryption) key.* Fortunately, the random oracle model enables a solution: to decrypt a ciphertext $\langle \bar{c}_1, \bar{c}_2 \rangle$ (where no prior decryption query was made using the same initial component $\bar{c}_1$), we generate a random key $\bar{k}$ and return the message $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$. We then *implicitly* set $H(\bar{r}) = \bar{k}$, where $\bar{r} \stackrel{\text{def}}{=} [c_1^{1/e} \bmod N]$. (Note that $\bar{r}$ may be unknown at this time, and we do not know how to compute it without the factorization of N.)

The only "catch" is that we must ensure consistency with both prior and later queries of $\mathcal{A}$ to the random oracle. But this is relatively simple:

- When decrypting, we first check for any prior random oracle query $H(\bar{r})$ such that $\bar{c}_1 = r^e \bmod N$ (and, if so, use for k the value previously returned in response to the previous random oracle query $H(\bar{r})$).
- When answering a random oracle query $H(\bar{r})$, we compute $\bar{c}_1 := [\bar{r}^e \bmod N]$ and check whether any previous decryption query used $\bar{c}_1$ as the first component of the ciphertext (and, if so, return the value $\bar{k}$ that was previously used to answer this prior decryption oracle query).

Actually, a simple data structure handles both cases: 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 $(\hat{r}, \hat{c}_1, \hat{k})$ with $\hat{c}_1 = [\hat{r}^e \bmod N]$. This entry means that we have defined $H(\hat{r}) = \hat{k}$.
- The second type of entry has the form $(\star, \hat{c}_1, \hat{k})$, which means that the value $\hat{r} \stackrel{\text{def}}{=} [\hat{c}_1^{1/e} \bmod N]$ is not yet known. (Again, we are not able to compute this value without the factorization of N.) An entry of this sort indicates that we are implicitly setting $H(\bar{r}) = \hat{k}$; in particular, when answering a decryption oracle query $\langle \bar{c}_1, \bar{c}_2 \rangle$ by $\mathcal{A}$, we return $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$. If $\mathcal{A}$ ever asks the random oracle query $H(\bar{r})$ we will return the correct answer $\hat{k}$ because we will check the table for any entry having $[\bar{r}^e \bmod N]$ as its second component.

We implement the above ideas as the following algorithm $\mathcal{A}'$:

Algorithm $\mathcal{A}'(N, e, c_1)$

1. Choose random $k \in \{0,1\}^n$. Triples $(\cdot, \cdot, \cdot)$ are stored in a table that initially contains only $(\star, c_1, k)$.
2. Run $\mathcal{A}$ on input $pk := \langle N, e \rangle$. When $\mathcal{A}$ makes a query $\langle \bar{c}_1, \bar{c}_2 \rangle$ to the decryption oracle, answer it as follows:
   - If there is an entry in the table whose second component is $\bar{c}_1$, let $\bar{k}$ be the third component of this entry. (That is, the entry is either of the form $(\bar{r}, \bar{c}_1, \bar{k})$ with $\bar{r}^e = \bar{c}_1 \bmod N$, or of the form $(\star, \bar{c}_1, \bar{k})$.) Return $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$.
   - Otherwise, choose a random $\bar{k} \leftarrow \{0,1\}^n$, return $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$, and store $(\star, \bar{c}_1, \bar{k})$ in the table.

   When $\mathcal{A}$ makes a query $\hat{r}$ to the random oracle, compute $\bar{c}_1 := [\bar{r}^e \bmod N]$ and answer the query as follows:
   - If there is an entry of the form $(\bar{r}, \bar{c}_1, \bar{k})$ in the table, return $\bar{k}$.

- If there is an entry of the form $(\star, \bar{c}_1, \bar{k})$ in the table, return $\bar{k}$ and store $(\bar{r}, \bar{c}_1, \bar{k})$ in the table.
- Otherwise, choose random $\bar{k} \leftarrow \{0,1\}^n$, return $\bar{k}$, and store $(\bar{r}, \bar{c}_1, \bar{k})$ 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 $c_2 := \mathsf{Enc}'_k(m_b)$. Give to $\mathcal{A}$ the ciphertext $\langle c_1, c_2 \rangle$, 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 (r, c_1, k) then output r.

Algorithm $\mathcal{A}'$ exactly carries out the strategy outlined earlier, with the only addition being that a random key k is chosen at the beginning of the experiment and $\mathcal{A}'$ implicitly sets $H([c_1^{1/e} \bmod N]) = k$.

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 $c_1 \leftarrow \mathbb{Z}_N^*$ 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 $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. 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. ■

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 the ciphertext expansion is relatively significant when very short messages are encrypted.

The *optimal asymmetric encryption padding* (OAEP) technique eliminates this drawback, and the ciphertext includes 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 (as 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[2] $n/2$ as a string $\hat{m}$ of length $2n$. 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 $H(x) \stackrel{\text{def}}{=} \bar{H}(1x)$.

In RSA-OAEP, encryption of a message m relative to a public key $\langle N, e\rangle$ (with $\|N\| > 2n$) is done by encoding m as $\hat{m}$ and then computing the ciphertext $c := [\hat{m}^d \bmod N]$. To decrypt, the receiver recovers $\hat{m}$ using its private key, and then reverses the encoding to recover the message m. A full description is given as Construction 13.9.

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.

Key generation. On input 1^n, run $\mathsf{GenRSA}(1^{n+1})$ to obtain (N, e, d) with $\|N\| > 2n$.[a] The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.

Encryption. To encrypt message $m \in \{0,1\}^{n/2}$ with respect to the public key $\langle N, e\rangle$, first choose random $r \leftarrow \{0,1\}^n$. Set $m' := m\|0^{n/2}$, compute $\hat{m}_1 := G(r) \oplus m'$, and then set

$$\hat{m} := \hat{m}_1 \,\|\, (r \oplus H(\hat{m}_1))$$

and interpret $\hat{m}$ as an element of $\mathbb{Z}_N^*$ in the natural way. Output the ciphertext $c := [\hat{m}^e \bmod N]$.

Decryption. To decrypt ciphertext c using private key $\langle N, d\rangle$, 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'.

[a] This explains our unusual choice to run GenRSA with input 1^{n+1} rather than input 1^n.

The RSA-OAEP encryption scheme.

[2] This matches Construction 13.9, but can be generalized for other message/encoding lengths.

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. A proof of security for the above construction is beyond the scope of the 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 secure for certain types of public exponents e (including the common case when $e = 3$). Variants of OAEP suitable for use with arbitrary public exponents or, more generally, with other trapdoor permutations, are also known; see the references at the end of this chapter.

13.3 RSA 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 constructions of digital signatures. The *full-domain hash* (FDH) signature scheme is perhaps the simplest to analyze. Though this, too, may be instantiated with any trapdoor permutation, we once again describe the RSA-FDH scheme which is based on RSA.

CONSTRUCTION 13.10

Let GenRSA be as in the previous sections, and let $H : \{0,1\}^* \to \{0,1\}^{2n}$ be a function.

Key generation. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.

Signing. To sin message $m \in \{0,1\}^*$ using the secret key $\langle N, d\rangle$, compute

$$\sigma := [H(m)^d \bmod N].$$

Verification. Given a signature σ on a message m with respect to the public key $\langle N, e\rangle$, output 1 iff $\sigma^e \stackrel{?}{=} H(m) \bmod N$.

The RSA-FDH signature scheme.

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 $\sigma := m^d \bmod N$. (As usual, the private key is $\langle N, d\rangle$ where (N, e, d) were output by some algorithm GenRSA.) Textbook

RSA is completely insecure, and in particular was shown in Section 12.3.1 to be vulnerable to the following attacks:

- An adversary can choose 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, it is possible to 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; we can now see why the attacks do not apply when H is modeled as a random oracle.

- Given σ, it is hard to find an m such that $H(m) = [\sigma^e \bmod N]$. (See, e.g., 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 $m = [m_1 \cdot m_2 \bmod N]$ since there is no reason to believe that $H(m_1 \cdot m_2) = H(m_1) \cdot H(m_2) \bmod N$. (And 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, say.

On the face of it, RSA-FDH may seem like an exact instantiation of the "hash-and-sign" paradigm (Section 12.4) using the textbook RSA signature scheme. The crucial difference is that in Section 12.4 we showed that the "hash-and-sign" paradigm converts a signature scheme Π that handles "short" messages into a signature scheme Π' that handles messages of arbitrary length, when the hash function being used is *collision resistant* and the original signature scheme Π is *existentially unforgeable under an adaptive chosen-message attack*. In contrast, here we are converting a *completely insecure scheme* Π (namely, textbook RSA signatures) into a *secure* scheme Π', but only by modeling the hash function as a *random oracle*.

We prove below that RSA-FDH is existentially unforgeable under an adaptive chosen-message attack, under the RSA assumption 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 only 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.

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 one query (chosen at random from among the q random 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 (in particular) 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 it attacks the 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, and hence 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 chosen by $\mathcal{A}$. 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 some known value σ. Note that 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 $\mathcal{A}'$.

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, 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

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n) = 1].$$

For the reader's convenience, we describe the steps of experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n)$:

1. *A random function H is chosen.*
2. $\mathsf{GenRSA}(1^n)$ *is run to generate* (N, e, d)*. 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.*
3. *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* $\sigma^e = H(m)$*, 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 $\mathcal{A}'(N, e, y^*)$

1. Choose $j \leftarrow \{1, \ldots, q\}$.
2. Run $\mathcal{A}$ on input 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 furthermore it will be the case that $\sigma_i^e = y_i \bmod N$. When $\mathcal{A}$ makes its ith random oracle query $H(m_i)$, answer it as follows:
   - If $i = j$, return y^*.
   - Otherwise, choose random $\sigma_i \leftarrow \mathbb{Z}_N^*$, 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[3] $m = m_i$ and answer the query as follows:
   - 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.
3. 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 polynomial time. Say the input to $\mathcal{A}'$ is generated by running $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and then choosing $y^* \leftarrow$

[3]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.

$\mathbb{Z}_N^*$ 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 output of $\mathcal{A}$. 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 $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n)$. 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^*, a value chosen at random from $\mathbb{Z}_N^*$.
- Queries $H(m_i)$ with $i \neq j$ are answered with $y_i = [\sigma_i^e \bmod N]$; since σ_i is chosen uniformly at random and RSA is a permutation, this means that y_i is uniformly distributed 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. 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, the 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. ∎

References and Additional Reading

The first formal treatment of the random oracle model was given by Bellare and Rogaway [19], though the idea of using a "random-looking" function in cryptographic applications had been suggested previously, most notably by [55]. Proper instantiation of a random oracle based on concrete cryptographic hash functions is discussed in [19, 20, 21, 63, 40].

The seminal negative result concerning the random oracle model is given by Canetti et al. [34], who show (contrived) schemes that can be proven secure in the random oracle model but demonstrably insecure for *any* concrete instantiation of the random oracle.

OAEP was introduced by Bellare and Rogaway [20] 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 [31, 59, 116] for further details.

The full-domain hash signature scheme was proposed by Bellare are Rogaway in their original paper on the random oracle model [19]. Later improvements include [21, 38, 39, 63], 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. Let Π be a signature scheme that is secure in the standard model. Construct a signature scheme Π_y where signing is done 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 Let $\Pi' = (\mathsf{Enc}', \mathsf{Dec}')$ be a private-key encryption scheme, and modify Construction 13.1 so that encryption is done as follows: To encrypt a message $m \in \{0,1\}^{\ell(n)}$ with respect to the public key $\langle N, e\rangle$, choose random $r \leftarrow \mathbb{Z}_N^*$, set $k := H(r)$, and output the ciphertext

$$\left\langle [r^e \bmod N],\ \mathsf{Enc}'_k(m) \right\rangle.$$

(Decryption is done in the obvious way.) Prove that if the RSA problem is hard relative to GenRSA, Π' has indistinguishable encryptions under a chosen-plaintext attack, and H is modeled as a random oracle, this modified construction is a CPA-secure public-key encryption scheme.

Index of Common Notation

Within each category, notation is given in order of appearance.

General notation:

- If A is a randomized algorithm, then $A(x)$ denotes running A on input x with a uniformly-chosen random tape. $A(x;r)$ denotes running A on input x using random tape r (note that this is a deterministic computation)
- $\wedge$ 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)
- $\Pr[X]$ denotes the probability of event X occurring
- $\{0,1\}^n$ is the set of all binary strings of length n
- $\{0,1\}^{<n}$ is the set of all binary strings of length less than n
- $\{0,1\}^*$ is the set of all finite strings of any length
- 0^n denotes the string comprised of n zeroes
- 1^n denotes the string comprised of n ones
- $\|x\|$ denotes the length of the string x, in bits. If x is a positive integer, this denotes the length of the binary representation of x when written with leading bit 1
- $\mathcal{O}(\cdot), \Theta(\cdot)$ see Section A.2
- $x\|y$ denotes the concatenation of two strings x and y
- $(x_1, \ldots, x_\ell)$ denotes a list (that may sometimes be referred to as a single value, for example as input to a function
- $x \leftarrow S$ denotes the process of choosing an element from S with the uniform distribution and letting x be the result
- $x \stackrel{\text{def}}{=} Y$ is used to denote that x is defined to be equal to Y

- $x := Y$ means that the variable x is assigned the value Y

Number theory:

- $\mathbb{Z}$ denotes the set of integers
- $a \mid b$ means a divides b
- $a \not\mid b$ 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 $0 \leq [a \bmod b] < b$.
- $x_1 = x_2 = \ldots = x_i \bmod N$ means that $x_1, \ldots, x_n$ are all congruent modulo 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)
- $\phi(N)$ equals the number of integers in $\{1, \ldots, N-1\}$ that are relatively prime to 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
- $\langle g \rangle$ 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 $+1$
- $\log x$ denotes the (standard) logarithm of x base 2 (this is the default)
- $\log_g h$ denotes the *discrete* logarithm of h to the base g; see Chapter 7

Crypto-specific notation:

- n is the security parameter
- (pk, sk) denotes the public/private key-pair (for public-key encryption and digital signatures)
- $\mathcal{A}$ denotes the adversary
- $\mathcal{A}^{\mathcal{O}(\cdot)}$ denotes the algorithm $\mathcal{A}$ with oracle access to $\mathcal{O}$ (essentially meaning that $\mathcal{A}$ may use external subroutine calls to $\mathcal{O}$ during its computation
- $\mathsf{Expt}^{\mathcal{A}}_{\text{name}}(n)$ is used to denote an experiment used for defining security. The experiment is defined by "name", for an adversary $\mathcal{A}$ and with security parameter n.
- k typically denotes a private key (as in private-key encryption and MACs)
- negl denotes a negligible function; that is, a function for which for every polynomial $p(\cdot)$ there exists an integer N such that for every $n > N$ it holds that $\mu(n) < 1/p(n)$. (It is typically used as $\mathsf{negl}(n)$ which means a function that is negligible in the security parameter n.)
- $\mathsf{poly}(n)$ denotes a polynomial; that is, $f(n) = \mathsf{poly}(n)$ should be read as "there exists a polynomial p such that $f(n) = \mathcal{O}(p(n))$
- $\mathsf{polylog}(n)$ denotes $\mathsf{poly}(\log(n))$
- IV denotes the initialization vector (used for block cipher modes of operation and collision-resistant hash functions)

Algorithms and procedures:

- $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ denote the key-generation, encryption and decryption procedures, respectively
- $(\mathsf{Gen}, \mathsf{MAC}, \mathsf{Vrfy})$ denote the key-generation, MAC generation and MAC verification procedures, respectively
- $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ denote the key-generation, signature generation and signature verification procedures, respectively

- (Gen, H) denote the key-generation and collision-resistant hash functions, respectively
- $\mathsf{GenPrime}(1^n)$ denotes a procedure for choosing a random prime of length n
- $\mathsf{GenModulus}(1^n)$ denotes a procedure for choosing two random primes p and q of length n each and outputting $N = pq$
- GenRSA denotes the procedure of generating RSA keys
- $\mathcal{G}(1^n)$ denotes a procedure that outputs the description of a group, a generator of that group and its order (which is typically an n-bit integer)
- G denotes a pseudorandom generator
- F_k denotes a pseudorandom function or permutation, keyed by k

References

[1] *Advances in Cryptology — Crypto '84*, volume 196 of *Lecture Notes in Computer Science*. Springer, 1985.

[2] *Advances in Cryptology — Crypto '86*, volume 263 of *Lecture Notes in Computer Science*. Springer, 1987.

[3] *Advances in Cryptology — Crypto '89*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[4] *Advances in Cryptology — Crypto '91*, volume 576 of *Lecture Notes in Computer Science*. Springer, 1992.

[5] *Advances in Cryptlolgy — Crypto '98*, volume 1462 of *Lecture Notes in Computer Science*. Springer, 1998.

[6] *Advances in Cryptology — Eurocrypt '99*, volume 1592 of *Lecture Notes in Computer Science*. Springer, 1999.

[7] *Advances in Cryptology — Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*. Springer, 2001.

[8] *Advances in Cryptology — Crypto 2005*, volume 3621 of *Lecture Notes in Computer Science*. Springer, 2005.

[9] ISO/IEC 9797. Data cryptographic techniques — data integrity mechanism using a cryptographic check function employing a block cipher algorithm, 1989.

[10] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addision Wesley, 2nd edition, 2002.

[11] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*.

[12] A. Akavia, S. Goldwasser, and S. Safra. Proving hard-core predicates using list decoding. In *Proc. 44th Annual Symposim on Foundations of Computer Science*, pages 146–157. IEEE, 2003.

[13] 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*, 17:194–209, 1988.

[14] 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.

[15] 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, 1997.

[16] 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* [5], pages 26–45.

[17] 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.

[18] 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.

[19] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM Conf. on Computer and Communications Security*, pages 62–73. ACM, 1993.

[20] 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.

[21] 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.

[22] 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.

[23] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[24] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[25] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Advances in Cryptology — Crypto '98* [5], pages 1–12.

[26] M. Blum. Coin flipping by telephone. In *Proc. IEEE COMPCOM*, pages 133–137, 1982.

[27] M. Blum and S. Goldwasser. An efficient probabilistic public-key encryption scheme which hides all partial information. In *Advances in Cryptology — Crypto '84* [1], pages 289–302.

[28] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[29] 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.

[30] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999.

[31] D. Boneh. Simplified OAEP for the RSA and Rabin functions. In *Advances in Cryptology — Crypto 2001* [7], pages 275–291.

[32] 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.

[33] D. Bressoud. *Factorization and Primality Testing*. Springer, 1989.

[34] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[35] D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In *Advances in Cryptology — Crypto '91* [4], pages 470–484.

[36] L.N. Childs. *A Concrete Introduction to Higher Algebra*. Undergraduate Texts in Mathematics. Springer, 2nd edition, 2000.

[37] D. Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, 1994.

[38] 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.

[39] 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.

[40] 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* [8], pages 430–448.

[41] 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.

[42] 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.

[43] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer, 2nd edition, 2005.

[44] J. Daemen and V. Rijmen. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer, 2002.

[45] W. Dai. Crypto++ 5.2.1 benchmarks. Available at `http://www.cryptopp.com/benchmarks.html`.

[46] 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.

[47] I. Damgård. A design principle for hash functions. In *Advances in Cryptology — Crypto '89* [3], pages 416–427.

[48] J. DeLaurentis. A further weakness in the common modulus protocol for the RSA cryptoalgorithm. *Cryptologia*, 8:253–259, 1984.

[49] M. Dietzfelbinger. *Primality Testing in Polynomial Time*. Springer, 2004.

[50] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[51] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

[52] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.

[53] 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.

[54] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, 1973.

[55] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — Crypto '86* [2], pages 186–194.

[56] 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.

[57] S. Fluhrer, I. Mantin, and A. Shamir. Attacks on RC4 and WEP. *CryptoBytes*, 5(2):26–34, 2002.

[58] J.B. Fraleigh. *A First Course in Abstract Algebra*. Addison Wesley, 7th edition, 2002.

[59] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology — Crypto 2001* [7], pages 260–274.

[60] T. El Gamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Info. Theory*, 31(4):469–472, 1985.

[61] C.F. Gauss. Disquisitiones arithmeticae (english edition). 1986.

[62] R. Gennaro, S. Halevi, and T. Rabin. Secure hash-and-sign signatures without the random oracle. In *Advances in Cryptology — Eurocrypt '99* [6], pages 123–139.

[63] E.-J. Goh, S. Jarecki, J. Katz, and Nan Wang. Efficient signature schemes with tight security reductions to the Diffie-Hellman problems. *J. Cryptology*, to appear.

[64] O. Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In *Advances in Cryptology — Crypto '86* [2], pages 104–110.

[65] O. Goldreich. *Foundations of Cryptography, vol. 1: Basic Tools*. Cambridge University Press, 2001.

[66] O. Goldreich. *Foundations of Cryptography, vol. 2: Basic Applications*. Cambridge University Press, 2004.

[67] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807", year = 1986.

[68] O. Goldreich, S. Goldwasser, and S. Micali. On the cryptographic applications of random functions. In *Advances in Cryptology — Crypto '84* [1], pages 276–288.

[69] 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, 1989.

[70] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[71] 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.

[72] 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.

[73] D. Hankerson, A.J. Menezes, and S.A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[74] J. Håstad. Solving simultaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2):336–341, 1988.

[75] 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.

[76] J. Håstad and M. Näslund. The security of all RSA and discrete log bits. *Journal of the ACM*, 51(2):187–230, 2004.

[77] I.N. Herstein. *Abstract Algebra*. Wiley, 3rd edition, 1996.

[78] H. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3):189–221, 2002. Available at `http://citeseer.ist.psu.edu/443539.html`.

[79] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996.

[80] A. Kalai. Generating random factored numbers, easily. *Journal of Cryptology*, 16(4):287–289, 2003.

[81] J. Katz. *Digital Signatures*. Springer, 2007.

[82] 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`.

[83] J. Katz and M. Yung. Characterization of security notions for probabilistic private-key encryption. *Journal of Cryptology*, 19(1):67–96, 2006.

[84] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 2nd edition, 2002.

[85] L. Kohnfelder. Towards a practical public-key cryptosystem, 1978. Undergraduate thesis, MIT.

[86] H. Krawczyk. The order of encryption and authentication for protecting communication (or: How secure is SSL?). In *Advances in Cryptology — Crypto 2001* [7], pages 310–331.

[87] H. Kugel. America's code breaker. Available for download from: `http://militaryhistory.about.com/`.

[88] L. Lamport. Constructing digital signatures from a one-way function. Technical Report CSL-98, SRI International, 1978.

[89] S. Levy. *Crypto: How the Code Rebels Beat the Government — Saving Privacy in the Digital Age*. Viking, 2001.

[90] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996.

[91] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[92] M. Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology — Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*. Springer, 1994.

[93] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRS Press, 1997.

[94] 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.

[95] R. Merkle. A certified digital signature. In *Advances in Cryptology — Crypto '89* [3], pages 218–238.

[96] R. Merkle. One way hash functions and DES. In *Advances in Cryptology — Crypto '89* [3], pages 428–446.

[97] G.L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[98] 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.

[99] 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, 1990.

[100] V.I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

[101] A.M. Odlyzko. Discrete logarithms: The past and the future. *Designs, Codes, and Cryptography*, 19(2/3):129–145, 2000.

[102] National Bureau of Standards. Data encryption standard (DES), 1977. Federal Information Processing Standard (FIPS), publication 46.

[103] National Bureau of Standards. DES modes of operation, 1980. Federal Information Processing Standard (FIPS), publication 81.

[104] National Institute of Standards and Technology. The keyed-hash message authentication code (HMAC), 2002. Federal Information Processing Standard (FIPS), publication 198.

[105] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology — Eurocrypt '99* [6], pages 223–238.

[106] 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.

[107] M.O. Rabin. Digitalized signatures as intractable as factorization. Technical Report TR-212, MIT/LCS, 1979.

[108] M.O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[109] C. Rackoff and D. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology — Crypto '91* [4], pages 433–444.

[110] 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.

[111] 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, 1990.

[112] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, 2nd edition, 1995.

[113] C.E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[114] 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.

[115] V. Shoup. Why chosen ciphertext security matters. Technical Report RZ 3076, IBM Zurich, November 1998. Available at `http://shoup.net/papers/expo.pdf`.

[116] V. Shoup. OAEP reconsidered. In *Advances in Cryptology — Crypto 2001* [7], pages 239–259.

[117] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2005. Also available at `http://www.shoup.net/ntb`.

[118] J.H. Silverman and J. Tate. *Rational Points on Elliptic Curves*. Undergraduate Texts in Mathematics. Springer, 1994.

[119] G. Simmons. A 'weak' privacy protocol using the RSA crypto algorithm. *Cryptologia*, 7:180–182, 1983.

[120] G. Simmons. A survey of information authentication. In G. Simmons, editor, *Contemporary Cryptology: The Science of Information Integrity*, pages 379–419. IEEE Press, 1992.

[121] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1), pages = "84–85", year = "1977").

[122] D.R. Stinson. Universal hashing and authentication codes. *Designs, Codes, and Cryptography*, 4(4):369–380, 1994.

[123] D.R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, 1st edition, 1995.

[124] D.R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, 3rd edition, 2005.

[125] W. Trappe and L. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2nd edition, 2005.

[126] 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.

[127] S.S. Wagstaff, Jr. *Cryptanalysis of Number Theoretic Ciphers*. Chapman & Hall/CRC Press, 2003.

[128] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology — Crypto 2005* [8], pages 17–36.

[129] 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.

[130] L. Washington. *Elliptic Curves: Number Theory and Cryptography*. Chapman & Hall/CRC Press, 2003.

[131] P. Weadon. The battle of Midway: AF is short of water, 2000. NSA Historical Publications. Available at: `http://www.nsa.gov` under Historical Publications.

[132] H. Wu. The misuse of RC4 in Microsoft Word and Excel. Available at `http://eprint.iacr.org/2005/007`.

[133] ANSI X9.9. American national standard for financial institution message authentication (wholesale), 1981.

[134] 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.

Appendix A

Mathematical Background

A.1 Identities and Inequalities

We list some standard identities and inequalities that are used 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, $f(n) = o(n^5)$.
- $f(n) = \Omega(n^3 \log n)$.
- $f(n) = \Theta(n^4)$.

♢

A.3 Basic Probability

We assume students are 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 $\Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2]$.

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

$$\Pr[F] = \sum_i \Pr[F \wedge E_i].$$

A special case is when $n = 2$ and $E_2 = \bar{E}_1$, giving

$$\Pr[F] = \Pr[F \wedge E_1] + \Pr[F \wedge \bar{E}_1].$$

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)

$$\Pr[E_1 \vee E_2] \leq \Pr[E_1] + \Pr[E_2].$$

Note that repeated application of the union bound gives

$$\Pr\left[\bigvee_{i=1}^{k} E_i\right] \leq \sum_{i=1}^{k} \Pr[E_i]$$

for any events $E_1, \ldots, E_k$.

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 $\Pr[E_2] \neq 0$. (In case $\Pr[E_2] = 0$ then we leave $\Pr[E_1 \mid E_2]$ undefined.) This represents the probability that event E_1 occurs once it is given that E_2 has occurred. It follows immediately from the definition that

$$\Pr[E_1 \wedge E_2] = \Pr[E_1 \mid E_2] \cdot \Pr[E_2]\,;$$

note that 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 $\mathsf{coll}(q, N)$. Determining the value of $\mathsf{coll}(q, N)$ has been extensively studied, and is related to the so-called *birthday problem*: 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, useful when q is small, 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, we have

$$\Pr[\mathsf{NoColl}_q] = \Pr[\mathsf{NoColl}_1]\cdot\Pr[\mathsf{NoColl}_2 \mid \mathsf{NoColl}_1]\cdots\Pr[\mathsf{NoColl}_q \mid \mathsf{NoColl}_{q-1}].$$

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 Section 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 Section 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 entire 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.) The final section of the appendix, 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, and is included mainly for completeness and reference.

Since our goal is only to establish that these 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). Similarly, 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 discussed in this Appendix are encapsulated in the following theorems:

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. *computing the sum $a+b$ and the difference $a-b$;*
2. *computing the product ab*
3. *computing positive integers q and $r < b$ such that $a = qb + r$ (i.e.,* division with remainder*);*
4. *computing the greatest common divisor of a and b,* $\gcd(a,b)$*;*
5. *computing integers X, Y with $Xa + Yb = \gcd(a,b)$.*

THEOREM B.2 (Modular Operations) *Given integers a, b, and N, it is possible to perform the following operations in time polynomial in $\|a\|$, $\|b\|$, and $\|N\|$:*

1. *computing the modular reduction $a \bmod N$;*
2. *computing the sum $(a+b) \bmod N$, the difference $(a-b) \bmod N$, and the product $ab \bmod N$;*
3. *computing the multiplicative inverse $a^{-1} \bmod N$, when a is invertible modulo N (it is also possible to determine whether a is invertible modulo N is time polynomial in n);*
4. *computing the exponentiation $a^b \bmod N$;*
5. *choosing an element uniformly at random from $\mathbb{Z}_N$ or $\mathbb{Z}_N^*$.*[1]

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 $\mathsf{poly}(\|b\|)$ group operations.*

Theorem B.2(4) is just a special case of the above.

[1] Of course, we need a probabilistic algorithm in this case. Furthermore, the algorithm has a small probability of failure; see Section B.2.4 for detailed discussion.

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 $\mathbb{Z}_N$.*

A algorithm with analogous properties exists for $\mathbb{Z}_N^*$ as well.

By way of notation, we let $x \leftarrow \mathbb{Z}_N$ denote random selection of an element x from $\mathbb{Z}_N$ using, e.g., the algorithm guaranteed by the above theorem (with analogous notation with $\mathbb{Z}_N^*$). Since the probability of outputting fail is negligible, we ignore it (and instead leave this possibility implicit).

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* 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* 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 i^{th} 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, 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* 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 $a \bmod b$ can both be done in time polynomial in $\|a\|$ and $\|b\|$, this implies that the entire algorithm runs in polynomial time.

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 $\mathsf{GCD}(b, a \bmod b)$

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, 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* $\mathsf{GCD}(a, b)$*, there are at most* $2\,\|b\| - 2$ *recursive calls to* GCD*.*

PROOF Let a_i, b_i $(i = 1, \ldots, \ell)$ denote the arguments to the i^{th} 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 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, called the *extended Euclidean algorithm*, can be used to find X, Y in addition to computing $\gcd(a, b)$. You are asked to show correctness of the

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 $(d, Y, X - Yq)$

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 (and not, e.g., a product of two primes) 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 often 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

$$[X \bmod N] \cdot a = Xa = 1 - YN = 1 \bmod N.$$

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} (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

More challenging is the case 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 first 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 simply 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 observation, 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 $a \in \mathbb{Z}_N$; exponent $b > 0$
Output: $a^b \bmod N$

```
x := 1
for i = 1 to b:
    x := x · a mod 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 can see that any algorithm depending on this recurrence is going to require $\Theta(b)$ time. We can do better by making use of the following recurrence:

$$a^b \bmod N = \left(a^{\frac{b}{2}}\right)^2 \bmod N \qquad \text{when } b \text{ is even}$$
$$a^b \bmod N = a \cdot \left(a^{\frac{b-1}{2}}\right)^2 \bmod N \qquad \text{when } b \text{ is odd.}$$

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. 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 done 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 $\|N\|$. In fact, looking carefully at the algorithm we see that it performs at most $2 \cdot \|b\|$ multiplications-plus-reductions modulo N.

For reasons of efficiency, an iterative algorithm is preferred to a recursive algorithm such as the one shown above; 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 widely believed to be hard when a and N are

ALGORITHM B.13
Algorithm ModExp **for efficient modular exponentiation**

Input: Modulus N; base $a \in \mathbb{Z}_N$; exponent $b > 0$
Output: $a^b \bmod N$

if $b = 1$ **return** a
else
  if b is even
    $t :=$ ModExp $(N, a, b/2)$
    return $t^2 \bmod N$
  if b is odd
    $t :=$ ModExp $(N, a, (b-1)/2)$
    return $a \cdot t^2 \bmod N$

chosen appropriately. Inverting the modular exponentiation function is known as solving the *discrete logarithm problem*, something we will discuss in great 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.

Considering the (additive) group $\mathbb{Z}_N$, note that this gives a different method for computing the "exponentiation"

$$b \cdot g \bmod N \stackrel{\text{def}}{=} \underbrace{g + \cdots + g}_{b \text{ times}} \bmod N$$

than 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 (the set) $\mathbb{Z}_N$. 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 will see further 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-random.") We first treat the problem in the abstract, 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) then 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.[2] That is:

ALGORITHM B.14
Choosing a random group element

Input: A (description of a) group $\mathbb{G}$; length-parameter ℓ; a 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,

[2] An alternate possibility is to output some arbitrary element of $\mathbb{G}$. Since we will eventually set t such that the probability that all t iterations fail is negligible, it does not matter much which method is used.

it outputs a uniformly-selected 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. More formally, we now 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 also that all groups sharing a particular value of n have the same length-parameter $\ell = \ell(n)$.)

As an example, we will later 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.

Actually, there is a trade-off between the running time of the algorithm and the probability that the algorithm outputs fail, since increasing t decreases the probability of failure but increases the worst-case running time. What we need for cryptographic applications is an algorithm where the worst-case running time is polynomial in n, while the failure probability is negligible in n. We claim that to achieve an algorithm of this sort, 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 first condition is obvious, since our algorithm for choosing a random element needs to check whether $x \in \mathbb{G}$. As for the second, we prove here merely that it is *sufficient*. Say the second condition holds, i.e., there is a polynomial p such that for every group $\mathbb{G}$ (in the given class) 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 some arbitrary class of groups. Instead, they must be verified for each particular class of interest.

The Case of $\mathbb{Z}_N$

Consider the class of 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 an 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 that results from the preceding discussion:

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 $\frac{\phi(N)}{N} > 1/2n$.*

(We note for completeness that 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, or when N is a product of two close-to-equal-length (distinct) primes.

The analysis is easy when N is prime. In this case $\phi(N) = N - 1$ and (as when we analyzed the algorithm for choosing a random element of $\mathbb{Z}_N$) we have

$$\frac{\phi(N)}{2^n} = \frac{N-1}{2^n} \geq \frac{2^{n-1}}{2^n} = \frac{1}{2}\,.$$

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 the rest of the 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. (Note also that choosing a random element from

$\mathbb{Z}_N$ is equivalent to selecting a random integer in the range $\{0, \ldots, N-1\}$.) 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 (or an x that is not in the desired range) 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. Note that, in contrast to most of the rest of the book, 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 $q > 1$, and note that this means that 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 $x \in \{0, \ldots, q-1\}$ the element $h = g^x$ is a generator of $\mathbb{G}$ exactly when $\gcd(x, q) = 1$. We have seen the set $\{x \in \{0, \ldots, q-1\} \mid \gcd(x, q) = 1\}$ 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 $\mathbb{G}$ iff*

$$h^{q_i} \neq 1 \quad \textit{for } i = 1, \ldots, k.$$

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 $q' \,|\, q$. 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) $h^{q_j} = h^{[q_j \bmod q']} = h^0 = 1$. ∎

Note that 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 it is possible 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 h^{q/p_i} can be computed in polynomial time in each iteration, we need only show that the number of iterations k is polynomial. This must be 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 $k \leq \log_2 q$.

The above algorithm 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 guarantees that a generator will be found 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 possibilities here include:

- 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).

- 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 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 [117] 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 [117, Chapter 5].

A nice result by Kalai [80] 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.

Index

AES
    competition, 173
    design, 173–175
    security, 175
Asymmetric, *see* public-key
Asymptotic security, 57
Avalanche effect, 156

Birthday attack, 125
Block cipher, *see* pseudorandom permutation
Block ciphers
    CBC mode of operation, 96
    CTR mode of operation, 97
    ECB mode of operation, 95
    modes of operation, 95–100
    OFB mode of operation, 96

Chosen-plaintext attacks, 81
Collision resistant hash function
    birthday attack, 125
    constructions in practice, 129–131
    definition, 124
    Merkle-Damgård, 127
    security notions, 124
    syntax, 123
Computational indistinguishability, 221
Concrete security, 49
Confusion-diffusion paradigm, 154
Cryptographic hash function, *see* collision-resistant hash function

DES
    avalanche effect, 164
    design, 162–164
    double-DES, 171
    security, 168–170
    triple-DES, 172
Differential cryptanalysis, 169, 176
Diffie-Hellman
    key exchange protocol, 309
    man-in-the-middle attacks, 313
    public-key revolution, 306

Encryption, *see* private-key encryption, *see* public-key encryption
    attack scenarios, 8
    basic setting, 4
    basic syntax, 5

Feistel network, 160

Hard-core predicate
    definition, 187
    Goldreich-Levin, 190
Historical ciphers, 9–18
    Caesar, 9
    substitution, 11
    Vigenère, 14
HMAC, 135

Kerckhoffs' principle, 6
Key distribution center (KDC), 303

Linear cryptanalysis, 169, 176

Merkle-Damgård, 127
Message authentication code
    CBC-MAC, 119
    definition, 112
    syntax, 110

Negligible function, 56

Nested MAC, 133

One-time pad, *see* perfect secrecy
One-way functions, 182–186
    candidates, 185
    definition, 183
    families, 184
    necessary for cryptography, 220
    sufficient for private-key cryptography, 215
One-way permutations, 184

Perfect secrecy
    definition, 31
    key size limitations, 37
    one-time pad, 34
    Shannon's theorem, 38
    Vernam's cipher, 34
Polynomial-time computation, 54
Private-key cryptography
    key distribution difficulties, 301
Private-key encryption
    CCA-security, 100, 137
    CPA-security, 82
    eavesdropping security, 63
    multiple message security, 78
    probabilistic encryption, 79
    semantic security, 64
    syntax, 60
Private-key limitations
    key distribution center, 303
    multiple key difficulties, 302
Pseudorandom function
    construction, 210
    definition, 87
    use in encryption, 89
Pseudorandom generator
    construction, 201
    definition, 70
    increasing expansion factor, 203
    use in encryption, 72
Pseudorandom permutation
    construction, 213
    definition, 93

Security parameter, 50
Signature scheme
    definition, 403, 410
    syntax, 402
Stream cipher, *see* pseudorandom generator
    modes for multiple encryptions, 79
    synchronized mode, 79
    unsynchronized mode, 80
    use in encryption, 72
Substitution-permutation network, 154
Symmetric, *see* private-key

Vernam's cipher, *see* perfect secrecy