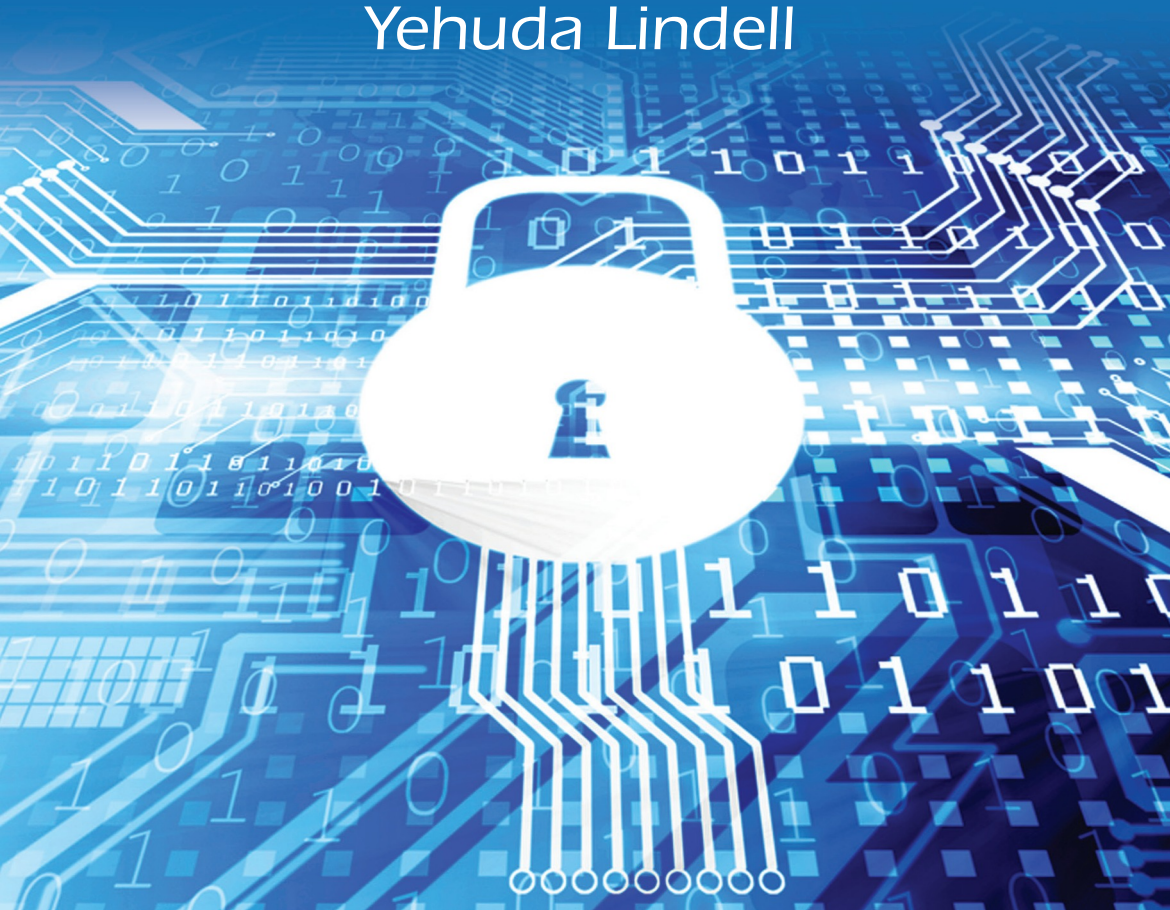
CHAPMAN & HALL/CRC
CRYPTOGRAPHY AND NETWORK SECURITY

INTRODUCTION TO MODERN CRYPTOGRAPHY

Second Edition

Jonathan Katz
Yehuda Lindell



CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

INTRODUCTION TO MODERN CRYPTOGRAPHY

Second Edition

CHAPMAN & HALL/CRC CRYPTOGRAPHY AND NETWORK SECURITY

Series Editor
Douglas R. Stinson

Published Titles

Lidong Chen and Guang Gong, Communication System Security

Shiu-Kai Chin and Susan Older, Access Control, Security, and Trust: A Logical Approach

M. Jason Hinek, Cryptanalysis of RSA and Its Variants

Antoine Joux, Algorithmic Cryptanalysis

Jonathan Katz and Yehuda Lindell, Introduction to Modern Cryptography, Second Edition

Sankar K. Pal, Alfredo Petrosino, and Lucia Maddalena, Handbook on Soft Computing for Video Surveillance

Burton Rosenberg, Handbook of Financial Cryptography and Security

Forthcoming Titles

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

CHAPMAN & HALL/CRC
CRYPTOGRAPHY AND NETWORK SECURITY

INTRODUCTION TO MODERN CRYPTOGRAPHY

Second Edition

Jonathan Katz

University of Maryland
College Park, MD, USA

Yehuda Lindell

Bar-Ilan University
Ramat Gan, Israel

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business

A CHAPMAN & HALL BOOK

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

Version Date: 20140915

International Standard Book Number-13: 978-1-4665-7027-6 (eBook - PDF)

Contents

Preface

The goal of our book remains the same as in the first edition: to present the basic paradigms and principles of modern cryptography to a general audience with a basic mathematics background. We have designed this book to serve as a textbook for undergraduate- or graduate-level courses in cryptography (in computer science, electrical engineering, 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—nor worked on revising it for the second edition—if the answer to that question were anything other than an unequivocal *yes*. What, in our opinion, distinguishes our book from other available books is that it provides a *rigorous* treatment of modern cryptography in an *accessible* manner appropriate for an introduction to the topic.

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 efficient constructions satisfying such strong definitions can be proven to exist (under rather mild assumptions).

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

- **The possibility of proofs of security:** The previous two principles serve as the basis for the idea that *cryptographic constructions can be proven secure* with respect to clearly stated definitions of security and relative to well-defined cryptographic assumptions. This concept is the essence of modern cryptography, and is what has transformed the field from an art to a science.

  The importance of this idea cannot be overemphasized. 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 advocates 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 principles of modern cryptography are relevant not only to the "theory of cryptography" community. The importance of precise definitions is, by now, widely understood and appreciated by developers and security engineers 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.

Changes in the Second Edition

In preparing the second edition, we have made a conscious effort to integrate a more practical perspective (without sacrificing a rigorous approach). This is reflected in a number of changes and additions we have made:

- We have increased our coverage of *stream ciphers*, introducing them as a variant of pseudorandom generators in Section 3.3.1, discussing stream-cipher modes of operation in Section 3.6.1, and describing modern stream-cipher design principles and examples in Section 6.1.

- We have emphasized the importance of *authenticated encryption* (see Section 4.5) and have added a section on secure communication sessions.

- We have moved our treatment of hash functions into its own chapter (Chapter 5), have included some standard applications of cryptographic hash functions (Section 5.6), and have added a section on hash-function design principles and widely used constructions (Section 6.3). We have also improved our treatment of birthday attacks (covering small-space birthday attacks in Section 5.4.2) and have added a discussion of rainbow tables and time/space tradeoffs (Section 5.4.3).

- We have included several important attacks on implementations of cryptography that arise in practice, including chosen-plaintext attacks on chained-CBC encryption (Section 3.6.2), padding-oracle attacks on CBC-mode encryption (Section 3.7.2), and timing attacks on MAC verification (Section 4.2).
- After much deliberation, we have decided to introduce the random-oracle model much earlier in the book (Section 5.5). This allows us to give a proper, integrated treatment of standardized, widely used public-key encryption and signature schemes in later chapters, instead of relegating them to second-class status in a chapter at the end of the book.
- We have strengthened our coverage of elliptic-curve cryptography (Section 8.3.4) and have added a discussion of its impact on recommended key lengths (Section 9.3).
- In the chapter on public-key encryption, we introduce the KEM/DEM paradigm as a form of hybrid encryption (see Section 11.3). We also cover DHIES/ECIES in addition to the RSA PKCS #1 standards.
- In the chapter on digital signatures, we now describe the construction of signatures from identification schemes using the Fiat–Shamir transform, with the Schnorr signature scheme as a prototypical example. We have also improved our coverage of DSA/ECDSA. We include brief discussions of SSL/TLS and signcryption, both of which serve as culminations of everything covered up to that point.
- In the "advanced topics" chapter, we have amplified our treatment of homomorphic encryption, and have included sections on secret sharing and threshold encryption.

Beyond the above, we have also edited the entire book to make extensive corrections as well as smaller adjustments, including more worked examples, to improve the exposition. Several additional exercises have also been added.

Guide to Using This Book

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

Required background. We have structured the book so that the only formal prerequisite is a course on discrete mathematics. Even here we rely on very little material: we assume familiarity with basic (discrete) probability and modular arithmetic. Students reading this book are also expected to have had some exposure to algorithms, mainly to be comfortable reading pseudocode and to be familiar with big-$\mathcal{O}$ notation. Many of these concepts are reviewed in Appendix A and/or when first used in the book.

Notwithstanding the above, the book does use definitions, proofs, and abstract 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, whether in an upper-level mathematics course or a course on discrete mathematics, algorithms, or computability theory.

Suggestions for course organization. The core material of this book, which we recommend should be covered in any introductory course on cryptography, consists of the following (in all cases, starred sections are excluded; more on this below):

- *Introduction and Classical Cryptography:* Chapters 1 and 2 discuss classical cryptography and set the stage for modern cryptography.

- *Private-Key (Symmetric) Cryptography:* Chapter 3 on private-key encryption, Chapter 4 on message authentication, and Chapter 5 on hash functions provide a thorough treatment of these topics.

  We also highly recommend covering Section 6.2, which deals with block-cipher design; in our experience students really enjoy this material, and it makes the abstract ideas they have learned in previous chapters more concrete. Although we do consider this core material, it is not used in the rest of the book and so can be safely skipped if desired.

- *Public-Key (Asymmetric) Cryptography:* Chapter 8 gives a self-contained introduction to all the number theory needed for the remainder of the book. The material in Chapter 9 is not used subsequently; however, we do recommend at least covering Section 9.3 on recommended key lengths. The public-key revolution is described in Chapter 10. Ideally, all of Chapters 11 and 12 should be covered; those pressed for time can pick and choose appropriately.

We are typically able to cover most of the above in a one-semester (35-hour) undergraduate course (omitting some proofs and skipping some topics, as needed) or, with some changes to add more material on theoretical foundations, in the first three-quarters of a one-semester graduate course. Instructors with more time available can proceed at a more leisurely pace or incorporate additional topics, as discussed below.

Those wishing to cover additional material, in either a longer course or a faster-paced graduate course, will find that the book is structured to allow flexible incorporation of other topics as time permits (and depending on the interests of the instructor). Specifically, the starred (*) sections and chapters may be covered in any order, or skipped entirely, without affecting the overall flow of the book. We have taken care to ensure that none of the core material depends on any of the starred material and, for the most part, the starred sections do not depend on each other. (When they do, this dependence is explicitly noted.)

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

- *Theory:* A more theoretically inclined course could include material from Section 3.2.2 (semantic security); Chapter 7 (one-way functions and hard-core predicates, and constructing pseudorandom generators, functions, and permutations from one-way permutations); Section 8.4 (one-way functions and collision-resistant hash functions from number-theoretic assumptions); Section 11.5.3 (RSA encryption without random oracles); and Section 12.6 (signatures without random oracles).

- *Mathematics:* A course directed at students with a strong mathematics background—or being taught by someone who enjoys this aspect of cryptography—could incorporate Section 4.6 (information-theoretic MACs in finite fields); some of the more advanced number theory from Chapter 8 (e.g., the Chinese remainder theorem and the Miller–Rabin primality test); and all of Chapter 9.

In either case, a selection of advanced topics from Chapter 13 could also be included.

Feedback and Errata

Our goal in writing this book was to make modern cryptography accessible to a wide audience beyond the "theoretical computer science" community. We hope you will let us know if we have succeeded. The many enthusiastic emails we have received in response to our first edition have made the whole process of writing this book worthwhile.

We are always happy to receive feedback. 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@biu.ac.il`; please put "Introduction to Modern Cryptography" in the subject line.

Acknowledgments

For the second edition: We are grateful to the many readers of the first edition who have sent us comments, suggestions, and corrections that helped to greatly improve the book. Discussions with Claude Crépeau, Bill Gasarch, Gene Itkis, Leonid Reyzin, Tom Shrimpton, and Salil Vadhan regarding the content and overall "philosophy" of the book were especially fruitful. We also thank Bar Alon, Gilad Asharov, Giuseppe Ateniese, Amir Azodi, Omer Berkman, Sergio de Biasi, Aurora Bristor, Richard Chang, Qingfeng Cheng, Kwan Tae Cho, Kyliah Clarkson, Ran Cohen, Nikolas Coukouma, Dana Dachman-Soled, Michael Fang, Michael Farcasin, Pooya Farshim, Marc Fischlin, Lance

Fortnow, Michael Fuhr, Bill Gasarch, Virgil Gligor, Carmit Hazay, Andreas Hübner, Karst Koymans, Eyal Kushilevitz, Steve Lai, Ugo Dal Lago, Armand Makowski, Tal Malkin, Steve Myers, Naveen Nathan, Ariel Nof, Eran Omri, Ruy de Queiroz, Eli Quiroz, Tal Rabin, Charlie Rackoff, Yona Raekow, Tzachy Reinman, Wei Ren, Ben Riva, Volker Roth, Christian Schaffner, Joachim Schipper, Dominique Schröder, Randy Shull, Nigel Smart, Christoph Sprenger, Aravind Srinivasan, John Steinberger, Aishwarya Thiruvengadam, Dave Tuller, Poorvi Vora, Avishai Yanai, Rupeng Yang, Arkady Yerukhimovich, Dae Hyun Yum, Hila Zarosim, and Konstantin Ziegler for their helpful corrections to the first edition and/or early drafts of the second edition.

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

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

Part I

Introduction and Classical Cryptography

Chapter 1

Introduction

1.1 Cryptography and Modern Cryptography

The *Concise Oxford English Dictionary* defines cryptography as "the art of writing or solving codes." This is historically accurate, but does not capture the current breadth of the field or its present-day scientific foundations. The definition focuses solely on the *codes* that have been used for centuries to enable secret communication. But cryptography nowadays encompasses much more than this: it deals with mechanisms for ensuring integrity, techniques for exchanging secret keys, protocols for authenticating users, electronic auctions and elections, digital cash, and more. Without attempting to provide a complete characterization, we would say that modern cryptography involves *the study of mathematical techniques for securing digital information, systems, and distributed computations against adversarial attacks.*

The dictionary definition also refers to cryptography as an *art*. Until late in the 20th century cryptography was, indeed, largely an art. Constructing good codes, or breaking existing ones, relied on creativity and a developed sense of how codes work. There was little theory to rely on and, for a long time, no working definition of what constitutes a good code. Beginning in the 1970s and 1980s, this picture of cryptography radically changed. A rich theory began to emerge, enabling the rigorous study of cryptography as a *science* and a mathematical discipline. This perspective has, in turn, influenced how researchers think about the broader field of computer security.

Another very important difference between classical cryptography (say, before the 1980s) and modern cryptography relates to its adoption. Historically, the major consumers of cryptography were military organizations and governments. Today, cryptography is everywhere! If you have ever authenticated yourself by typing a password, purchased something by credit card over the Internet, or downloaded a verified update for your operating system, you have undoubtedly used cryptography. And, more and more, programmers with relatively little experience are being asked to "secure" the applications they write by incorporating cryptographic mechanisms.

In short, cryptography has gone from a heuristic set of tools concerned with ensuring secret communication for the military to a science that helps secure systems for ordinary people all across the globe. This also means that cryptography has become a more central topic within computer science.

Goals of this book. Our goal is to make the basic principles of modern cryptography accessible to students of computer science, electrical engineering, or mathematics; to professionals who want to incorporate cryptography in systems or software they are developing; and to anyone with a basic level of mathematical maturity who is interested in understanding this fascinating field. After completing this book, the reader should appreciate the security guarantees common cryptographic primitives are intended to provide; be aware of standard (secure) constructions of such primitives; and be able to perform a basic evaluation of new schemes based on their proofs of security (or lack thereof) and the mathematical assumptions underlying those proofs. It is not our intention for readers to become experts—or to be able to design new cryptosystems—after finishing this book, but we have attempted to provide the terminology and foundational material needed for the interested reader to subsequently study more advanced references in the area.

This chapter. The focus of this book is the formal study of modern cryptography, but we begin in this chapter with a more informal discussion of "classical" cryptography. Besides allowing us to ease into the material, our treatment in this chapter will also serve to motivate the more rigorous approach we will be taking in the rest of the book. Our intention here is not to be exhaustive and, as such, this chapter should not be taken as a representative historical account. The reader interested in the history of cryptography is invited to consult the references at the end of this chapter.

1.2 The Setting of Private-Key Encryption

Classical cryptography was concerned with designing and using *codes* (also called *ciphers*) that enable two parties to communicate secretly in the presence of an eavesdropper who can monitor all communication between them. In modern parlance, codes are called *encryption schemes* and that is the terminology we will use here. Security of all classical encryption schemes relied on a secret—a *key*—shared by the communicating parties in advance and unknown to the eavesdropper. This scenario is known as the *private-key* (or *shared-/secret-key*) setting, and private-key encryption is just one example of a cryptographic primitive used in this setting. Before describing some historical encryption schemes, we discuss private-key encryption more generally.

In the setting of private-key encryption, two parties share a key and use this key when they want to communicate secretly. One party can send a message, or *plaintext*, to the other by using the shared key to *encrypt* (or "scramble") the message and thus obtain a *ciphertext* that is transmitted to the receiver. The receiver uses the same key to *decrypt* (or "unscramble") the ciphertext and recover the original message. Note the same key is used to convert the

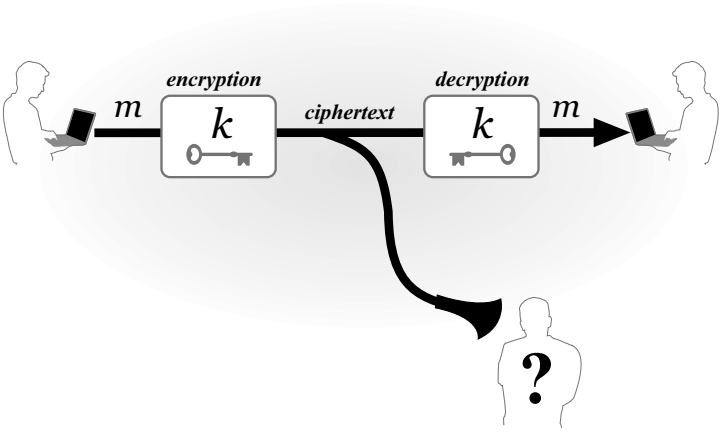


FIGURE 1.1: One common setting of private-key cryptography (here, encryption): two parties share a key that they use to communicate securely.

plaintext into a ciphertext and back; that is why this is also known as the *symmetric-key* setting, where the symmetry lies in the fact that both parties hold the same key that is used for encryption and decryption. This is in contrast to *asymmetric*, or *public-key*, encryption (introduced in Chapter 10), where encryption and decryption use different keys.

As already noted, the goal of encryption is to keep the plaintext hidden from an eavesdropper who can monitor the communication channel and observe the ciphertext. We discuss this in more detail later in this chapter, and spend a great deal of time in Chapters 2 and 3 formally defining this goal.

There are two canonical applications of private-key cryptography. In the first, there are two distinct parties separated in *space*, e.g., a worker in New York communicating with her colleague in California; see Figure 1.2. These two users are assumed to have been able to securely share a key in advance of their communication. (Note that if one party simply sends the key to the other over the public communication channel, then the eavesdropper obtains the key too!) Often this is easy to accomplish by having the parties physically meet in a secure location to share a key before they separate; in the example just given, the co-workers might arrange to share a key when they are both in the New York office. In other cases, sharing a key securely is more difficult. For the next several chapters we simply assume that sharing a key is possible; we will revisit this issue in Chapter 10.

The second widespread application of private-key cryptography involves the same party communicating with itself over *time*. (See Figure 1.2.) Consider, e.g., disk encryption, where a user encrypts some plaintext and stores the resulting ciphertext on their hard drive; the same user will return at a later

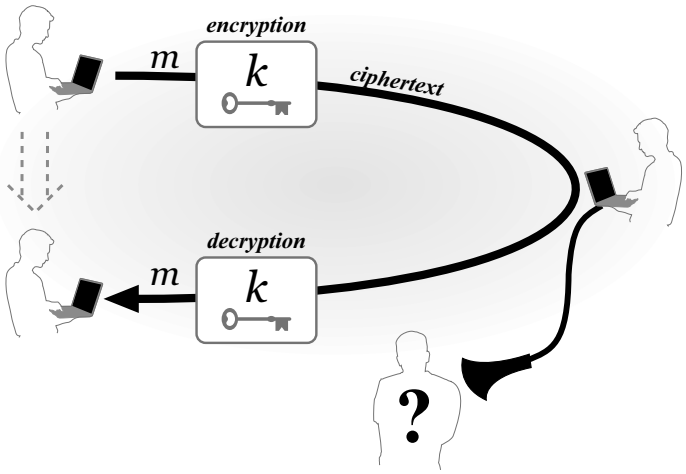


FIGURE 1.2: Another common setting of private-key cryptography (again, encryption): a single user stores data securely over time.

point in time to decrypt the ciphertext and recover the original data. The hard drive here serves as the communication channel on which an attacker might eavesdrop by gaining access to the hard drive and reading its contents. "Sharing" the key is now trivial, though the user still needs a secure and reliable way to remember/store the key for use at a later point in time.

The syntax of encryption. Formally, a private-key encryption scheme is defined by specifying a *message space* $\mathcal{M}$ along with three algorithms: a procedure for generating keys (Gen), a procedure for encrypting (Enc), and a procedure for decrypting (Dec). The message space $\mathcal{M}$ defines the set of "legal" messages, i.e., those supported by the scheme. The 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.

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

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

An encryption scheme must satisfy the following correctness requirement: for every key k output by Gen and every message $m \in \mathcal{M}$, it holds that

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

In words: encrypting a message and then decrypting the resulting ciphertext (using the same key) yields the original message.

The set of all possible keys output by the key-generation algorithm is called the *key space* and is denoted by $\mathcal{K}$. Almost always, Gen simply chooses a uniform key from the key space; in fact, one can assume without loss of generality that this is the case (see Exercise 2.1).

Reviewing our earlier discussion, an encryption scheme can be used by two parties who wish to communicate as follows. First, Gen is run to obtain a key k that the parties share. Later, when one party wants to send a plaintext m to the other, she computes $c := \mathsf{Enc}_k(m)$ and sends the resulting ciphertext c over the public channel to the other party.[1] 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, 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 any ciphertexts transmitted by those parties. It is for this reason that the communicating parties must share the key k securely and keep k completely secret from everyone else. Perhaps they should keep the decryption algorithm Dec secret, too? For that matter, might it not be better for them to keep all the details of the encryption scheme secret?

In the late 19th century, Auguste Kerckhoffs argued the opposite in a paper he wrote elucidating several design principles for military ciphers. One of the most important of these, now known simply as *Kerckhoffs' principle*, was:

> *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.*

That is, an encryption scheme should be designed to be secure *even if* an eavesdropper knows all the details of the scheme, so long as the attacker doesn't know the key being used. Stated differently, security should not rely on the encryption scheme being secret; instead, Kerckhoffs' principle demands that *security rely solely on secrecy of the key*.

There are three primary arguments in favor of Kerckhoffs' principle. The first is that it is significantly easier for the parties to maintain secrecy of a short key than to keep secret the (more complicated) algorithm they are using. This is especially true if we imagine using encryption to secure the communication between all pairs of employees in some organization. Unless each pair of parties uses their own, unique algorithm, some parties will know the algorithm used by others. Information about the encryption algorithm might be leaked by one of these employees (say, after being fired), or obtained by an attacker using reverse engineering. In short, it is simply unrealistic to assume that the encryption algorithm will remain secret.

[1] We use ":=" to denote deterministic assignment, and assume for now that Enc is deterministic. A list of common notation can be found in the back of the book.

Second, in case the honest parties' shared, secret information *is* ever exposed, it will be much easier for them to change a key than to replace an encryption scheme. (Consider updating a file versus installing a new program.) Moreover, it is relatively trivial to generate a new random secret, whereas it would be a huge undertaking to design a new encryption scheme.

Finally, for large-scale deployment it is significantly easier for users to all rely on the same encryption algorithm/software (with different keys) than for everyone to use their own custom algorithm. (This is true even for a single user who is communicating with several different parties.) In fact, it is desirable for encryption schemes to be *standardized* so that (1) compatibility is ensured by default and (2) users will utilize an encryption scheme that has undergone public scrutiny and in which no weaknesses have been found.

Nowadays Kerckhoffs' principle is understood as advocating that cryptographic designs be made completely public, in stark contrast to the notion of "security by obscurity" which suggests that keeping algorithms secret improves security. It is very dangerous to use a proprietary, "home-brewed" algorithm (i.e., a non-standardized algorithm designed in secret by some company). In contrast, published designs undergo public review and are therefore likely to be stronger. Many years of experience have demonstrated that it is very difficult to construct good cryptographic schemes. Therefore, our confidence in the security of a scheme is much higher if it has been extensively studied (by experts other than the designers of the scheme) and no weaknesses have been found. As simple and obvious as it may sound, the principle of open cryptographic design (i.e., Kerckhoffs' principle) has been ignored over and over again with disastrous results. Fortunately, today there are enough secure, standardized, and widely available cryptosystems that there is no reason to use anything else.

1.3 Historical Ciphers and Their Cryptanalysis

In our study of "classical" cryptography we will examine some historical encryption schemes and show that they are insecure. Our main aims in presenting this material are (1) to highlight the weaknesses of an "ad hoc" approach to cryptography, and thus motivate the modern, rigorous approach that will be taken in the rest of the book, and (2) to demonstrate that simple approaches to achieving secure encryption are unlikely to succeed. Along the way, we will present some central principles of cryptography inspired by the weaknesses of these historical schemes.

In this section, plaintext characters are written in `lower case` and ciphertext characters are written in `UPPER CASE` for typographical clarity.

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 CE:

> *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...*

Julius Caesar encrypted by shifting the letters of the alphabet 3 places forward: `a` was replaced with `D`, `b` with `E`, and so on. At the very end of the alphabet, the letters wrap around and so `z` was replaced with `C`, `y` with `B`, and `x` with `A`. For example, encryption of the message `begin the attack now`, with spaces removed, gives:

`EHJLQWKHDWWDFNQRZ`.

An immediate problem with this cipher is that the encryption method is *fixed*; there is no key. Thus, anyone learning how Caesar encrypted his messages would be able to decrypt effortlessly.

Interestingly, a variant of this cipher called ROT-13 (where the shift is 13 places instead of 3) is still used nowadays in various online forums. It is understood that this does not provide any cryptographic security; it 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. The *shift cipher* can be viewed as a keyed variant of Caesar's cipher.[2] Specifically, in the shift cipher the key k is a number between 0 and 25. To encrypt, letters are shifted as in Caesar's cipher, but now by k places. Mapping this to the syntax of encryption described earlier, the message space consists of arbitrary length strings of English letters with punctuation, spaces, and numerals removed, and with no distinction between upper and lower case. Algorithm Gen outputs a uniform key $k \in \{0, \ldots, 25\}$; algorithm Enc takes a key k and a plaintext and shifts each letter of the plaintext forward k positions (wrapping around at the end of the alphabet); and algorithm Dec takes a key k and a ciphertext and shifts every letter of the ciphertext *backward* k positions.

A more mathematical description is obtained by equating the English alphabet with the set $\{0, \ldots, 25\}$ (so $\mathtt{a} = 0$, $\mathtt{b} = 1$, etc.). The message space $\mathcal{M}$ is then any finite sequence of integers from this set. Encryption of the message $m = m_1 \cdots m_\ell$ (where $m_i \in \{0, \ldots, 25\}$) using key k is given by

$$\mathsf{Enc}_k(m_1 \cdots m_\ell) = c_1 \cdots c_\ell, \quad \text{where } c_i = [(m_i + k) \bmod 26].$$

(The notation $[a \bmod N]$ denotes the remainder of a upon division by N, with $0 \leq [a \bmod N] < N$. We refer to the process mapping a to $[a \bmod N]$

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

as *reduction modulo* N; we will have more to say about this beginning in Chapter 8.) Decryption of a ciphertext $c = c_1 \cdots c_\ell$ using key k is given by

$$\mathsf{Dec}_k(c_1 \cdots c_\ell) = m_1 \cdots m_\ell, \quad \text{where } m_i = [(c_i - k) \bmod 26].$$

Is the shift cipher secure? Before reading on, try to decrypt the following ciphertext that was generated using the shift cipher and a secret key k:

`OVDTHUFWVZZPISLRLFZHYLAOLYL`.

Is it possible to recover the message without knowing k? Actually, it is trivial! The reason is that there are only 26 possible keys. So one can try to decrypt the ciphertext using every possible key and thereby obtain a list of 26 candidate plaintexts. The correct plaintext will certainly be on this list; moreover, if the ciphertext is "long enough" then the correct plaintext will likely be the only candidate on the list that "makes sense." (The latter is not necessarily true, but will be true most of the time. Even when it is not, the attack narrows down the set of potential plaintexts to at most 26 possibilities.) By scanning the list of candidates it is easy to recover the original plaintext.

An attack that involves trying every possible key is called a *brute-force* or *exhaustive-search* attack. Clearly, for an encryption scheme to be secure it must not be vulnerable to such an attack.[3] This observation is known as the *sufficient key-space principle*:

> *Any secure encryption scheme must have a key space that is sufficiently large to make an exhaustive-search attack infeasible.*

One can debate what amount of effort makes a task "infeasible," and an exact determination of feasibility depends on both the resources of a potential attacker and the length of time the sender and receiver want to ensure secrecy of their communication. Nowadays, attackers can use supercomputers, tens of thousands of personal computers, or graphics processing units (GPUs) to speed up brute-force attacks. To protect against such attacks the key space must therefore be very large—say, of size at least 2^{70}, and even larger if one is concerned about long-term security against a well-funded attacker.

The sufficient key-space principle gives a *necessary* condition for security, but not a *sufficient* one. The next example demonstrates this.

The mono-alphabetic substitution cipher. In the shift cipher, the key defines a map from each letter of the (plaintext) alphabet to some letter of the (ciphertext) alphabet, where the map is a fixed shift determined by the key. In the *mono-alphabetic substitution cipher*, the key also defines a map on the alphabet, but the map is now allowed to be *arbitrary* subject only to the constraint that it be one-to-one so that decryption is possible. The key

[3] Technically, this is only true if the message space is larger than the key space; we will return to this point in Chapter 2. Encryption schemes used in practice have this property.

space thus consists of all *bijections*, or *permutations*, of the alphabet. So, for example, the key that defines the following permutation

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`. The name of this cipher comes from the fact that the key defines a (fixed) substitution for individual characters of the plaintext.

Assuming the English alphabet is being used, the key space is of size $26! = 26 \cdot 25 \cdot 24 \cdots 2 \cdot 1$, or approximately 2^{88}, and a brute-force attack is infeasible. This, however, does not mean the cipher is secure! In fact, as we will show next, it is easy to break this scheme even though it has a large key space.

Assume 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). The mono-alphabetic substitution cipher can then be attacked by utilizing statistical patterns of the English language. (Of course, the same attack works for any language.) The attack relies on the facts that:

1. For any key, 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 frequency distribution of individual letters in the English language is known (see Figure 1.3). Of course, very short texts may deviate from this distribution, but even texts consisting of only a few sentences tend to have distributions that are very close to the average.

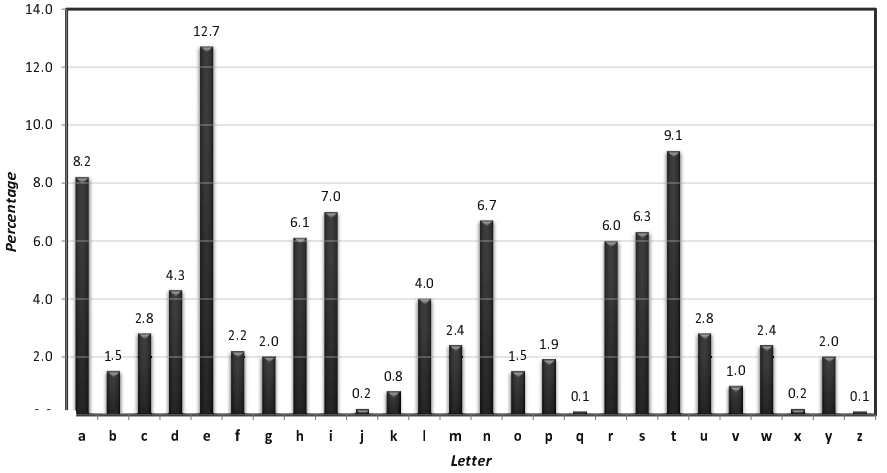


FIGURE 1.3: Average letter frequencies for English-language text.

The attack works by tabulating the frequency distribution of characters in the ciphertext, i.e., recording that `A` appeared 11 times, `B` appeared 4 times, and so on. These frequencies are then compared to the known letter frequencies of normal English text. One can then guess parts of the mapping defined by the key based on the observed frequencies. For example, since `e` is the most frequent letter in English, one can guess that the most frequent character in the ciphertext corresponds to the plaintext character `e`, and so on. Some of the guesses may be wrong, but enough of the guesses will be correct to enable relatively quick decryption (especially utilizing other knowledge of English, such as the fact that `u` generally follows `q`, and that `h` is likely to appear between `t` and `e`). We conclude that although the mono-alphabetic substitution cipher has a large key space, it is still insecure.

It should not be 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 ciphertext—this should convince you how easy the attack is to carry out. (Use Figure 1.3 to help you.)

```
JGRMQOYGHMVBJWRWQFPWHGFFDQGFPFZRKBEEBJIZQQOCIBZKLFAFGQVFZFWWE
OGWOPFGFHWOLPHLRLOLFDMFGQWBLWBWQOLKFWBYLBLYLFSFLJGRMQBOLWJVFP
FWQVHQWFFPQOQVFPQOCFPOGFWFJIGFQVHLHLROQVFGWJVFPFOLFHGQVQVFILE
OGQILHQFQGIQVVOSFAFGBWQVHQWIJVWJVFPFWHGFIWIHZZRQGBABHZQOCGFHX
```

An improved attack on the shift cipher. We can use letter-frequency tables to give an improved attack on the shift cipher. Our previous attack on the shift cipher required decrypting the ciphertext using each possible key, and then checking which key results in a plaintext that "makes sense." A drawback of this approach is that it is somewhat difficult to automate, since it is difficult for a computer to check whether a given plaintext "makes sense." (We do not claim that it would be impossible, as the attack could be automated using a dictionary of valid English words. We only claim that it would not be trivial to automate.) Moreover, there may be cases—we will see one later—where the plaintext characters are distributed just like English-language text even though the plaintext itself is not valid English, in which case checking for a plaintext that "makes sense" will not work.

We now describe an attack that does not suffer from these drawbacks. As before, associate the letters of the English alphabet with $0, \ldots, 25$. Let p_i, with $0 \leq p_i \leq 1$, denote the frequency of the ith letter in normal English text (ignoring spaces, punctuation, etc.). Calculation using Figure 1.3 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 frequency of the ith letter of the alphabet in this ciphertext; i.e., q_i is simply the number

of occurrences of the ith letter of the alphabet in the ciphertext divided by the length of the ciphertext. If the key is k, then p_i should be roughly equal to q_{i+k} for all i, because the ith letter is mapped to the $(i+k)$th letter. (We use $i+k$ instead of the more cumbersome $[i+k \bmod 26]$.) Thus, if we compute

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

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

The Vigenère (poly-alphabetic shift) cipher. The statistical attack on the mono-alphabetic substitution cipher can be carried out because the key defines a fixed mapping that is applied letter-by-letter to the plaintext. Such an attack could be thwarted by using a *poly-alphabetic substitution cipher* where the key instead defines a mapping that is applied on *blocks* of plaintext characters. Here, for example, a key might map the 2-character block `ab` to `DZ` while mapping `ac` to `TY`; note that the plaintext character `a` does not get mapped to a fixed ciphertext character. Poly-alphabetic substitution ciphers "smooth out" the frequency distribution of characters in the ciphertext and make it harder to perform statistical analysis.

The *Vigenère cipher*, a special case of the above also called the poly-alphabetic *shift* cipher, works by applying several independent instances of the shift cipher in sequence. The key is now viewed as a *string* of letters; encryption is done by shifting each plaintext character by the amount indicated by the next character of the key, wrapping around in the key when necessary. (This degenerates to the shift cipher if the key has length 1.) For example, encryption of the message `tellhimaboutme` using the key `cafe` would work as follows:

Plaintext:	`tellhimaboutme`
Key (repeated):	`cafecafecafeca`
Ciphertext:	`VEQPJIREDOZXOE`

(The key need not be an English word.) This is exactly the same as encrypting the first, fifth, ninth, ... characters with the shift cipher and key `c`; the second, sixth, tenth, ... characters with key `a`; the third, seventh, ... characters with `f`; and the fourth, eighth, ... characters with `e`. Notice that in the above example `l` is mapped once to `Q` and once to `P`. Furthermore, the ciphertext character `E` is sometimes obtained from `e` and sometimes from `a`. Thus, the character frequencies of the ciphertext are "smoothed out," as desired.

If the key is sufficiently long, cracking this cipher appears daunting. Indeed, it had been considered by many to be "unbreakable," and although it was invented in the 16th century, a systematic attack on the scheme was only devised hundreds of years later.

Attacking the Vigenère cipher. A first observation in attacking the Vigenère cipher is that *if the length of the key is known* then attacking the cipher is relatively easy. Specifically, say the length of the key, also called the *period*, is t. Write the key k as $k = k_1 \cdots k_t$ where each k_i is a letter of the alphabet. An observed ciphertext $c = c_1 c_2 \cdots$ can be divided into t parts where each part can be viewed as having been encrypted using a shift cipher. Specifically, for all $j \in \{1, \ldots, t\}$ the ciphertext characters

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

all resulted by shifting the corresponding characters of the plaintext by k_j positions. We refer to the above sequence of characters as the jth *stream*. All that remains is to determine, for each of the t streams, which of the 26 possible shifts was used. This is not as trivial as in the case of the shift cipher, because it is no longer possible to simply try different shifts in an attempt to determine when decryption of a stream "makes sense." (Recall that a stream does not correspond to consecutive letters of the plaintext.) Furthermore, trying to guess the entire key k at once would require a brute-force search through 26^t different possibilities, which is infeasible for large t. Nevertheless, we can still use letter-frequency analysis to analyze each stream independently. Namely, for each stream we tabulate the frequency of each ciphertext character and then check which of the 26 possible shifts yields the "right" probability distribution for that stream. Since this can be carried out independently for each stream (i.e., for each character of the key), this attack takes time $26 \cdot t$ rather than time 26^t.

A more principled, easier-to-automate approach is to use the improved method for attacking the shift cipher discussed earlier. That attack did not rely on checking for a plaintext that "made sense," but only relied on the underlying frequency distribution of characters in the plaintext.

Either of the above approaches gives a successful attack when the key length is known. What if the key length is unknown?

Note first that as long as the maximum length T of the key is not too large, we can simply repeat the above attack T times (for each possible value $t \in \{1, \ldots, T\}$). This leads to at most T different candidate plaintexts, among which the true plaintext will likely be easy to identify. So an unknown key length is not a serious obstacle.

There are also more efficient ways to determine the key length from an observed ciphertext. One is to use *Kasiski's method*, published in the mid-19th century. The first step here is to identify repeated patterns of length 2 or 3 in the ciphertext. These are likely the result of certain bigrams or trigrams that appear frequently in the plaintext. For example, consider the common word "`the`." This word will be mapped to different ciphertext characters, depending on its position in the plaintext. However, if it appears twice in the same relative position, then it will be mapped to the same ciphertext characters. For a sufficiently long plaintext, there is thus a good chance that "`the`" will be mapped repeatedly to the same ciphertext characters.

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

Plaintext:	`the man and the woman retrieved the letter from the post office`
Key:	`bea dsb ead sbe adsbe adsbeadsb ead sbeads bead sbe adsb eadsbe`
Ciphertext:	`ULE PSO ENG LII WREBR RHLSMEYWE XHH DFXTHJ GVOP LII PRKU SFIADI`

The word `the` is mapped sometimes to `ULE`, sometimes to `LII`, and sometimes to `XHH`. However, it is mapped *twice* to `LII`, and in a long enough text it is likely that it would be mapped multiple times to each of these possibilities. Kasiski's observation was that the distance between such repeated appearances (assuming they are not coincidental) must be a multiple of the period. (In the above example, the period is 5 and the distance between the two appearances of `LII` is 30, which is 6 times the period.) Therefore, the greatest common divisor of the distances between repeated sequences (assuming they are not coincidental) will yield the key length t or a multiple thereof.

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

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

in the first stream all resulted from encryption 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 *in some shifted order*. In more detail: let q_i denote the observed frequency of the ith English letter in this stream; this is simply the number of occurrences of the ith letter of the alphabet divided by the total number of letters in the stream. If the shift used here is j (i.e., if the first character k_1 of the key is equal to j), then for all i we expect $q_{i+j} \approx p_i$, where p_i is the frequency of the ith letter of the alphabet in standard English text. (Once again, we use q_{i+j} in place of $q_{[i+j \bmod 26]}$.) But this means that the sequence $q_0, \ldots, q_{25}$ is just the sequence $p_0, \ldots, p_{25}$ shifted j places. As a consequence (cf. Equation (1.1)):

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

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

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

When $\tau = t$ we expect $S_\tau \approx 0.065$, as discussed above. On the other hand, if τ is not a multiple of t we expect that all characters will occur with roughly equal

probability in the sequence $c_1, c_{1+\tau}, c_{1+2\tau}, \ldots$, and so we expect $q_i \approx 1/26$ for all i. In this case we will obtain

$$S_\tau \approx \sum_{i=0}^{25} \left(\frac{1}{26}\right)^2 \approx 0.038.$$

The smallest value of τ for which $S_\tau \approx 0.065$ is thus likely the key length. One can further validate a guess τ by carrying out a similar calculation using the second stream $c_2, c_{2+\tau}, c_{2+2\tau}, \ldots$, etc.

Ciphertext length and cryptanalytic attacks. The above attacks on the Vigenère cipher require a longer ciphertext than the attacks on previous schemes. For example, the index of coincidence method requires c_1, c_{1+t}, c_{1+2t} (where t is the actual key length) to be sufficiently long in order to ensure that the observed frequencies match what is expected; the ciphertext itself must then be roughly t times larger. Similarly, the attack we showed on the mono-alphabetic substitution cipher requires a longer ciphertext than the attack on the shift cipher (which can work for encryptions of even a single word). This illustrates that a longer key can, in general, require the cryptanalyst to obtain more ciphertext in order to carry out an attack. (Indeed, the Vigenère cipher can be shown to be secure if the key is as long as what is being encrypted. We will see a similar phenomenon in the next chapter.)

Conclusions. We have presented only a few historical ciphers. Beyond their historical interest, our aim in presenting them was to illustrate some important lessons. Perhaps the most important is that *designing secure ciphers is hard*. The Vigenère cipher remained unbroken for a long time. Far more complex schemes have also been used. But a complex scheme is not necessarily secure, and all historical schemes have been broken.

1.4 Principles of Modern Cryptography

As should be clear from the previous section, cryptography was historically more of an art than a science. Schemes were designed in an ad hoc manner and evaluated based on their perceived complexity or cleverness. A scheme would be analyzed to see if any attacks could be found; if so, the scheme would be "patched" to thwart that attack, and the process repeated. Although there may have been agreement that some schemes were *not* secure (as evidenced by an especially damaging attack), there was no agreed-upon notion of what requirements a "secure" scheme should satisfy, and no way to give evidence that any specific scheme was secure.

Over the past several decades, cryptography has developed into more of a science. Schemes are now developed and analyzed in a more systematic

manner, with the ultimate goal being to give a rigorous *proof* that a given construction is secure. In order to articulate such proofs, we first need *formal definitions* that pin down exactly what "secure" means; such definitions are useful and interesting in their own right. As it turns out, most cryptographic proofs rely on currently unproven *assumptions* about the algorithmic hardness of certain mathematical problems; any such assumptions must be made explicit and be stated precisely. An emphasis on definitions, assumptions, and proofs distinguishes *modern* cryptography from classical cryptography; we discuss these three principles in greater detail in the following sections.

1.4.1 Principle 1 – Formal Definitions

One of the key contributions of modern cryptography has been the recognition that formal definitions of security are *essential* for the proper design, study, evaluation, and usage of cryptographic primitives. Put bluntly:

> *If you don't understand what you want to achieve, how can you possibly know when (or if) you have achieved it?*

Formal definitions provide such understanding by giving a clear description of what threats are in scope and what security guarantees are desired. As such, definitions can help guide the design of cryptographic schemes. Indeed, it is much better to formalize what is required *before* the design process begins, rather than to come up with a definition *post facto* once the design is complete. The latter approach risks having the design phase end when the designers' patience is exhausted (rather than when the goal has been met), or may result in a construction achieving *more* than is needed at the expense of efficiency.

Definitions also offer a way to evaluate and analyze what is constructed. With a definition in place, one can study a proposed scheme to see if it achieves the desired guarantees; in some cases, one can even *prove* a given construction secure (see Section 1.4.3) by showing that it meets the definition. On the flip side, definitions can be used to conclusively show that a given scheme is *not* secure, insofar as the scheme does not satisfy the definition. In particular, note that the attacks in the previous section do not automatically demonstrate that any of the schemes shown there is "insecure." For example, the attack on the Vigenère cipher assumed that sufficiently long English text was being encrypted, but could the Vigenère cipher be "secure" if short English text, or compressed text (which will have roughly uniform letter frequencies), is encrypted? It is hard to say without a formal definition in place.

Definitions enable a meaningful comparison of schemes. As we will see, there can be multiple (valid) ways to define security; the "right" one depends on the context in which a scheme is used. A scheme satisfying a weaker definition may be more efficient than another scheme satisfying a stronger definition; with precise definitions we can properly evaluate the trade-offs between the two schemes. Along the same lines, definitions enable secure usage of schemes. Consider the question of deciding which encryption scheme

to use for some larger application. A sound way to approach the problem is to first understand what notion of security is required for that application, and then find an encryption scheme satisfying that notion. A side benefit of this approach is *modularity*: a designer can "swap out" one encryption scheme and replace it with another (that also satisfies the necessary definition of security) without having to worry about affecting security of the overall application.

Writing a formal definition forces one to think about what is essential to the problem at hand and what properties are extraneous. Going through the process often reveals subtleties of the problem that were not obvious at first glance. We illustrate this next for the case of encryption.

An example: secure encryption. A common mistake is to think that formal definitions are not needed, or are trivial to come up with, because "everyone has an intuitive idea of what security means." This is not the case. As an example, we consider the case of encryption. (The reader may want to pause here to think about how they would formally define what it means for an encryption scheme to be secure.) Although we postpone a formal definition of secure encryption to the next two chapters, we describe here informally what such a definition should capture.

In general, a security definition has two components: a security guarantee (or, from the attacker's point of view, what constitutes a successful attack on the scheme) and a threat model. The security guarantee defines what the scheme is intended to prevent the attacker from doing, while the threat model describes the power of the adversary, i.e., what actions the attacker is assumed able to carry out.

Let's start with the first of these. What should a secure encryption scheme guarantee? Here are some thoughts:

- *It should be impossible for an attacker to recover the key.* We have previously observed that if an attacker can determine the key shared by two parties using some scheme, then that scheme cannot be secure. However, it is easy to come up with schemes for which key recovery is impossible, yet the scheme is blatantly insecure. Consider, e.g., the scheme where $\mathsf{Enc}_k(m) = m$. The ciphertext leaks no information about the key (and so the key cannot be recovered if it is long enough) yet the message is sent in the clear! We thus see that inability to recover the key is not sufficient for security. This makes sense: the aim of encryption is to protect the *message*; the key is a means for achieving this but is, in itself, unimportant.

- *It should be impossible for an attacker to recover the entire plaintext from the ciphertext.* This definition is better, but is still far from satisfactory. In particular, this definition would consider an encryption scheme secure if its ciphertexts revealed 90% of the plaintext, as long as 10% of the plaintext remained hard to figure out. This is clearly unacceptable in most common applications of encryption; for example, when encrypting

a salary database, we would be justifiably upset if 90% of employees' salaries were revealed!

- *It should be impossible for an attacker to recover any character of the plaintext from the ciphertext.* This looks like a good definition, yet is still not sufficient. Going back to the example of encrypting a salary database, we would not consider an encryption scheme secure if it reveals whether an employee's salary is more than or less than $100,000, even if it does not reveal any particular digit of that employee's salary. Similarly, we would not want an encryption scheme to reveal whether employee A makes more than employee B.

  Another issue is how to formalize what it means for an adversary to "recover a character of the plaintext." What if an attacker correctly guesses, through sheer luck or external information, that the least significant digit of someone's salary is 0? Clearly that should not render an encryption scheme insecure, and so any viable definition must somehow rule out such behavior as being a successful attack.

- *The "right" answer: regardless of any information an attacker already has, a ciphertext should leak no additional information about the underlying plaintext.* This informal definition captures all the concerns outlined above. Note in particular that it does not try to define what information about the plaintext is "meaningful"; it simply requires that *no* information be leaked. This is important, as it means that a secure encryption scheme is suitable for all potential applications in which secrecy is required.

  What is missing here is a precise, mathematical formulation of the definition. How should we capture an attacker's prior knowledge about the plaintext? And what does it mean to (not) leak information? We will return to these questions in the next two chapters; see especially Definitions 2.3 and 3.12.

Now that we have fixed a security *goal*, it remains to specify a *threat model*. This specifies what "power" the attacker is assumed to have, but does not place any restrictions on the adversary's *strategy*. This is an important distinction: we specify what we assume about the adversary's abilities, but we do *not* assume anything about *how it uses* those abilities. It is impossible to foresee what strategies might be used in an attack, and history has proven that attempts to do so are doomed to failure.

There are several plausible options for the threat model in the context of encryption; standard ones, in order of increasing power of the attacker, are:

- **Ciphertext-only attack:** This is the most basic attack, and refers to a scenario where the adversary just observes a ciphertext (or multiple ciphertexts) and attempts to determine information about the underlying plaintext (or plaintexts). This is the threat model we have been

implicitly assuming when discussing classical encryption schemes in the previous section.

- **Known-plaintext attack:** Here, the adversary is able to learn one or more plaintext/ciphertext pairs generated using some key. The aim of the adversary is then to deduce information about the underlying plaintext of some *other* ciphertext produced using the same key.

  All the classical encryption schemes we have seen are trivial to break using a known-plaintext attack; we leave a demonstration as an exercise.

- **Chosen-plaintext attack:** In this attack, the adversary can obtain plaintext/ciphertext pairs (as above) for plaintexts *of its choice*.

- **Chosen-ciphertext attack:** The final type of attack is one where the adversary is additionally able to obtain (some information about) the *decryption* of ciphertexts of its choice, e.g., whether the decryption of some ciphertext chosen by the attacker yields a valid English message. The adversary's aim, once again, is to learn information about the underlying plaintext of some *other* ciphertext (whose decryption the adversary is unable to obtain directly).

None of these threat models is inherently better than any other; the right one to use depends on the environment in which an encryption scheme is deployed.

The first two types of attack are the easiest to carry out. In a ciphertext-only attack, the only thing the adversary needs to do is eavesdrop on the public communication channel over which encrypted messages are sent. In a known-plaintext attack it is assumed that the adversary somehow also obtains ciphertexts corresponding to known plaintexts. This is often easy to accomplish 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 reports secret until their release date; in this case, anyone eavesdropping on the ciphertext will later obtain the corresponding plaintext.

In the latter two attacks the adversary is assumed to be able to obtain encryptions and/or decryptions of plaintexts/ciphertexts of its choice. This may at first seem strange, and we defer a more detailed discussion of these attacks, and their practicality, to Section 3.4.2 (for chosen-plaintext attacks) and Section 3.7 (for chosen-ciphertext attacks).

1.4.2 Principle 2 – Precise Assumptions

Most modern cryptographic constructions cannot be proven secure unconditionally; such proofs would require resolving questions in the theory of computational complexity that seem far from being answered today. The result of

this unfortunate state of affairs is that proofs of security typically rely on *assumptions*. Modern cryptography requires any such assumptions to be made explicit and mathematically precise. At the most basic level, this is simply because mathematical proofs of security require this. But there are other reasons as well:

1. *Validation of assumptions:* By their very nature, assumptions are statements that are not proven but are instead conjectured to be true. In order to strengthen our belief in some assumption, it is necessary for the assumption to be studied. The more the assumption is examined and tested without being refuted, the more confident we are that the assumption is true. Furthermore, study of an assumption can provide 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, it cannot be studied and (potentially) refuted. Thus, a pre-condition to increasing our confidence in an assumption is having a precise statement of what exactly is being assumed.

2. *Comparison of schemes:* Often in cryptography we are presented with two schemes that can both be proven to satisfy some definition, each based on a different assumption. Assuming all else is equal, which scheme should be preferred? If the assumption on which the first scheme is based is *weaker* than the assumption on which the second scheme is based (i.e., the second assumption implies the first), then the first scheme is preferable 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 not comparable, then the general rule is to prefer the scheme that is based on the better-studied assumption in which there is greater confidence.

3. *Understanding the necessary assumptions:* An encryption scheme may be based on some underlying building block. If some weaknesses are later found in the building block, how can we tell whether the encryption scheme is still secure? If the underlying assumptions regarding the building block are made clear as part of proving security of the scheme, then we need only check whether the required assumptions are affected by the new weaknesses that were found.

A question that sometimes arises is: rather than prove a scheme secure based on some other assumption, why not simply assume that the construction *itself* is secure? In some cases—e.g., when a scheme has successfully resisted attack for many years—this may be a reasonable approach. But this approach is never preferred, and is downright dangerous when a new scheme is being introduced. The reasons above help explain why. First, an assumption that has been tested for several years is preferable to a new, ad hoc assumption

that is introduced along with a new construction. Second, there is a general preference for assumptions that are simpler to state, since such assumptions are easier to study and to (potentially) refute. So, for example, an assumption that some mathematical problem is hard to solve is simpler to study and evaluate than the assumption that an encryption scheme satisfies a complex security definition. Another advantage of relying on "lower-level" assumptions (rather than just assuming a construction is secure) is that these low-level assumptions can typically be used in other constructions. Finally, low-level assumptions can provide *modularity*. Consider an encryption scheme whose security relies on some assumed property of one of its building blocks. If the underlying building block turns out *not* to satisfy the stated assumption, the encryption scheme can still be instantiated using a different component that is believed to satisfy the necessary requirements.

1.4.3 Principle 3 – Proofs of Security

The two principles described above allow us to achieve our goal of providing a rigorous *proof* that a construction satisfies a given definition under certain specified assumptions. Such proofs are especially important in the context of cryptography where there is an attacker who is actively trying to "break" some scheme. Proofs of security give an iron-clad guarantee—relative to the definition and assumptions—that no attacker will succeed; this is much better than taking an unprincipled or heuristic approach to the problem. Without a proof that no adversary with the specified resources can break some scheme, we are left only with our intuition that this is the case. Experience has shown that intuition in cryptography and computer security is disastrous. There are countless examples of unproven schemes that were broken, sometimes immediately and sometimes years after being developed.

Summary: Rigorous vs. Ad Hoc Approaches to Security

Reliance on definitions, assumptions, and proofs constitutes a rigorous approach to cryptography that is distinct from the informal approach of classical cryptography. Unfortunately, unprincipled, "off-the-cuff" solutions are still designed and deployed by those wishing to obtain a quick solution to a problem, or by those who are simply unknowledgable. We hope this book will contribute to an awareness of the rigorous approach and its importance in developing provably secure schemes.

1.4.4 Provable Security and Real-World Security

Much of modern cryptography now rests on sound mathematical foundations. But this does not mean that the field is no longer partly an *art* as well. The rigorous approach leaves room for creativity in developing definitions suited to contemporary applications and environments, in proposing new

mathematical assumptions or designing new primitives, and in constructing novel schemes and proving them secure. There will also, of course, always be the art of *attacking* deployed cryptosystems, even if they are proven secure. We expand on this point next.

The approach taken by modern cryptography has revolutionized the field, and helps provide confidence in the security of cryptographic schemes deployed in the real world. But it is important not to overstate what a proof of security implies. A proof of security is always relative to the *definition* being considered and the *assumption(s)* being used. If the security guarantee does not match what is needed, or the threat model does not capture the adversary's true abilities, then the proof may be irrelevant. Similarly, if the assumption that is relied upon turns out to be false, then the proof of security is meaningless.

The take-away point is that provable security of a scheme does not necessarily imply security of that scheme in the real world.[4] While some have viewed this as a drawback of provable security, we view this optimistically as illustrating the *strength* of the approach. To attack a provably secure scheme in the real world, it suffices to focus attention on the definition (i.e., to explore how the idealized definition differs from the real-world environment in which the scheme is deployed) or the underlying assumptions (i.e., to see whether they hold). In turn, it is the job of cryptographers to continually refine their definitions to more closely match the real world, and to investigate their assumptions to test their validity. Provable security does not end the age-old battle between attacker and defender, but it does provide a framework that helps shift the odds in the defender's favor.

References and Additional Reading

In this chapter, we have studied just a few of the known historical ciphers. There are many others of both historical and mathematical interest, and we refer the reader to textbooks by Stinson [168] or Trappe and Washington [169] for further details. The important role cryptography has played throughout history is a fascinating subject covered in books by Kahn [97] and Singh [163].

Kerckhoffs' principles were elucidated in [103, 104]. Shannon [154] was the first to pursue a rigorous approach to cryptography based on precise definitions and mathematical proofs; we explore his work in the next chapter.

[4] Here we are not even considering the possibility of an incorrect *implementation* of the scheme. Poorly implemented cryptography is a serious problem in the real world, but this problem is somewhat outside the scope of cryptography *per se*.

Exercises

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

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

1.3 Provide a formal definition of the Gen, Enc, and Dec algorithms for the Vigenère cipher. (Note: there are several plausible choices for Gen; choose one.)

1.4 Implement the attacks described in this chapter for the shift cipher and the Vigenère cipher.

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

1.6 Assume an attacker knows that a user's password is either `abcd` or `bedg`. Say the user encrypts his password using the shift cipher, and the attacker sees the resulting ciphertext. Show how the attacker can determine the user's password, or explain why this is not possible.

1.7 Repeat the previous exercise for the Vigenère cipher using period 2, using period 3, and using period 4.

1.8 The shift, substitution, and Vigenère ciphers can also be defined over the 128-character ASCII alphabet (rather than the 26-character English alphabet).

(a) Provide a formal definition of each of these schemes in this case.

(b) Discuss how the attacks we have shown in this chapter can be modified to break each of these modified schemes.

Chapter 2

Perfectly Secret Encryption

In the previous chapter we presented historical encryption schemes and showed how they can be broken with little computational effort. In this chapter, we look at the other extreme and study encryption schemes that are *provably* secure even against an adversary with unbounded computational power. Such schemes are called *perfectly secret*. Besides rigorously defining the notion, we will explore conditions under which perfect secrecy can be achieved. (Beginning in this chapter, we assume familiarity with basic probability theory. The relevant notions are reviewed in Appendix A.3.)

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-1970s and 1980s, 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 are used and rigorous proofs are given, but it will not be necessary to rely on any unproven computational assumptions. It is clearly advantageous to avoid such assumptions; we will see, however, that doing so 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.

Beginning with this chapter, we will define security and analyze schemes using probabilistic experiments involving algorithms making randomized choices; a basic example is given by communicating parties' choosing a random key. Thus, before returning to the subject of cryptography *per se*, we briefly discuss the issue of generating randomness suitable for cryptographic applications.

Generating randomness. Throughout the book, we will simply assume that parties have access to an unlimited supply of independent, unbiased random bits. In practice, where do these random bits come from? In principle, one could generate a small number of random bits by hand, e.g., by flipping a fair coin. But such an approach is not very convenient, nor does it scale.

Modern *random-number generation* proceeds in two steps. First, a "pool" of high-entropy data is collected. (For our purposes a formal definition of entropy is not needed, and it suffices to think of entropy as a measure of unpredictability.) Next, this high-entropy data is processed to yield a sequence of nearly independent and unbiased bits. This second step is necessary since high-entropy data is not necessarily uniform.

For the first step, some source of unpredictable data is needed. There are several ways such data can be acquired. One technique is to rely on external inputs, for example, delays between network events, hard-disk access times, keystrokes or mouse movements made by the user, and so on. Such data is likely to be far from uniform, but if enough measurements are taken the resulting pool of data is expected to have sufficient entropy. More sophisticated approaches—which, by design, incorporate random-number generation more tightly into the system at the hardware level—have also been used. These rely on physical phenomena such as thermal/shot noise or radioactive decay. Intel has recently developed a processor that includes a digital random-number generator on the processor chip and provides a dedicated instruction for accessing the resulting random bits (after they have been suitably processed to yield independent, unbiased bits, as discussed next).

The processing needed to "smooth" the high-entropy data to obtain (nearly) uniform bits is a non-trivial one, and is discussed briefly in Section 5.6.4. Here, we just give a simple example to give an idea of what is done. Imagine that our high-entropy pool results from a sequence of *biased* coin flips, where "heads" occurs with probability p and "tails" with probability $1-p$. (We do assume, however, that the result of any coin flip is *independent* of all other coin flips. In practice this assumption is typically not valid.) The result of 1,000 such coin flips certainly has high entropy, but is not close to uniform. We can obtain a uniform distribution by considering the coin flips in pairs: if we see a head followed by a tail then we output "0," and if we see a tail followed by a head then we output "1." (If we see two heads or two tails in a row, we output nothing, and simply move on to the next pair.) The probability that any pair results in a "0" is $p \cdot (1-p)$, which is exactly equal to the probability that any pair results in a "1," and we thus obtain a uniformly distributed output from our initial high-entropy pool.

Care must be taken in how random bits are produced, and using poor random-number generators can often leave a good cryptosystem vulnerable to attack. One should use a random-number generator that is *designed for cryptographic use*, rather than a "general-purpose" random-number generator, which is not suitable for cryptographic applications. In particular, the `rand()` function in the C stdlib.h library is *not* cryptographically secure, and using it in cryptographic settings can have disastrous consequences.

2.1 Definitions

We begin by recalling and expanding upon 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 (finite) *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 (finite) *key space*, i.e., the set of all possible keys that can be output by Gen. 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 now allow the encryption algorithm to be probabilistic (so $\mathsf{Enc}_k(m)$ might output a different ciphertext when run multiple times), and 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 $c := \mathsf{Enc}_k(m)$. Looking ahead, we also sometimes use the notation $x \leftarrow S$ to denote uniform selection of x from a set S.) 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 $m \in \mathcal{M}$. We assume *perfect correctness*, meaning that for all $k \in \mathcal{K}$, $m \in \mathcal{M}$, and any ciphertext c output by $\mathsf{Enc}_k(m)$, it holds that $\mathsf{Dec}_k(c) = m$ with probability 1. Perfect correctness 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 to yield the message m.

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 the one defined by running Gen and taking the output. (It is almost always the case that Gen chooses a key uniformly from $\mathcal{K}$ and, in fact, we may assume this without loss of generality; see Exercise 2.1.) We let K be a random variable denoting the value of the key output by Gen; thus, for any $k \in \mathcal{K}$, $\Pr[K = k]$ denotes the probability that the key output by Gen is equal to k. Similarly, we let M be a random variable denoting the message being encrypted, so $\Pr[M = m]$ denotes the probability that the message takes on the value $m \in \mathcal{M}$. The probability distribution of the message is not determined by the encryption scheme itself, but instead reflects the likelihood of different messages being sent by the parties using the scheme, as well as an adversary's uncertainty about what will be sent. As an example, an adversary may know that the message will either be `attack today` or `don't attack`. 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 $\Pr[M = \texttt{attack today}] = 0.7$ and $\Pr[M = \texttt{don't attack}] = 0.3$.

K and M are assumed to be independent, i.e., what is being communicated by the parties is independent of the key they happen to share. This makes sense, among other reasons, because the distribution over $\mathcal{K}$ is determined by

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

the encryption scheme itself (since it is defined by Gen), while the distribution over $\mathcal{M}$ depends on the context in which the encryption scheme is being used.

Fixing an encryption scheme and a distribution over $\mathcal{M}$ determines a distribution over the space of ciphertexts $\mathcal{C}$ given by choosing a key $k \in \mathcal{K}$ (according to Gen) and a message $m \in \mathcal{M}$ (according to the given distribution), and then computing the ciphertext $c \leftarrow \mathsf{Enc}_k(m)$. We let C be the random variable denoting the resulting ciphertext and so, for $c \in \mathcal{C}$, write $\Pr[C = c]$ to denote the probability that the ciphertext is equal to the fixed value c.

Example 2.1

We work through a simple example for the shift cipher (cf. Section 1.3). Here, by definition, we have $\mathcal{K} = \{0, \ldots, 25\}$ with $\Pr[K = k] = 1/26$ for each $k \in \mathcal{K}$.

Say we are given the following distribution over $\mathcal{M}$:

$$\Pr[M = \mathtt{a}] = 0.7 \quad \text{and} \quad \Pr[M = \mathtt{z}] = 0.3.$$

What is the probability that the ciphertext is $\mathtt{B}$? There are only two ways this can occur: either $M = \mathtt{a}$ and $K = 1$, or $M = \mathtt{z}$ and $K = 2$. By independence of M and K, we have

$$\begin{aligned}\Pr[M = \mathtt{a} \wedge K = 1] &= \Pr[M = \mathtt{a}] \cdot \Pr[K = 1] \\ &= 0.7 \cdot \left(\frac{1}{26}\right).\end{aligned}$$

Similarly, $\Pr[M = \mathtt{z} \wedge K = 2] = 0.3 \cdot \left(\frac{1}{26}\right)$. Therefore,

$$\begin{aligned}\Pr[C = \mathtt{B}] &= \Pr[M = \mathtt{a} \wedge K = 1] + \Pr[M = \mathtt{z} \wedge K = 2] \\ &= 0.7 \cdot \left(\frac{1}{26}\right) + 0.3 \cdot \left(\frac{1}{26}\right) \;=\; 1/26.\end{aligned}$$

We can calculate conditional probabilities as well. For example, what is the probability that the message $\mathtt{a}$ was encrypted, given that we observe ciphertext $\mathtt{B}$? Using Bayes' Theorem (Theorem A.8) we have

$$\begin{aligned}\Pr[M = \mathtt{a} \mid C = \mathtt{B}] &= \frac{\Pr[C = \mathtt{B} \mid M = \mathtt{a}] \cdot \Pr[M = \mathtt{a}]}{\Pr[C = \mathtt{B}]} \\ &= \frac{0.7 \cdot \Pr[C = \mathtt{B} \mid M = \mathtt{a}]}{1/26}.\end{aligned}$$

Note that $\Pr[C = \mathtt{B} \mid M = \mathtt{a}] = 1/26$, since if $M = \mathtt{a}$ then the only way $C = \mathtt{B}$ can occur is if $K = 1$ (which occurs with probability $1/26$). We conclude that $\Pr[M = \mathtt{a} \mid C = \mathtt{B}] = 0.7$. ♢

Example 2.2

Consider the shift cipher again, but with the following distribution over $\mathcal{M}$:

$$\Pr[M = \mathtt{kim}] = 0.5,\ \Pr[M = \mathtt{ann}] = 0.2,\ \Pr[M = \mathtt{boo}] = 0.3.$$

What is the probability that $C = \texttt{DQQ}$? The only way this ciphertext can occur is if $M = \texttt{ann}$ and $K = 3$, or $M = \texttt{boo}$ and $K = 2$, which happens with probability $0.2 \cdot 1/26 + 0.3 \cdot 1/26 = 1/52$.

So what is the probability that `ann` was encrypted, conditioned on observing the ciphertext `DQQ`? A calculation as above using Bayes' Theorem gives $\Pr[M = \texttt{ann} \mid C = \texttt{DQQ}] = 0.4$. ♢

Perfect secrecy. We are now ready to define the notion of *perfect secrecy*. 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. This adversary also knows the encryption scheme being used; the only thing unknown to the adversary is the key shared by the parties. A message is chosen by one of the honest parties and encrypted, and the resulting ciphertext transmitted to the other party. The adversary can *eavesdrop* on the parties' communication, and thus observe this ciphertext. (That is, this is a ciphertext-only attack, where the attacker gets only a single ciphertext.) For a scheme to be perfectly secret, observing this ciphertext should have *no effect* on the adversary's knowledge regarding the actual message that was sent; in other words, the *a posteriori* probability that some message $m \in \mathcal{M}$ was sent, conditioned on the ciphertext that was observed, should be no different from the *a priori* probability that m would be sent. This means that the ciphertext reveals nothing about the underlying plaintext, and the adversary learns absolutely nothing about the plaintext that was encrypted. Formally:

DEFINITION 2.3 *An encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with 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.)

We now give an equivalent formulation of perfect secrecy. Informally, this formulation requires that the probability distribution of the ciphertext does not depend on the plaintext, i.e., for any two messages $m, m' \in \mathcal{M}$ the distribution of the ciphertext when m is encrypted should be identical to the distribution of the ciphertext when m' is encrypted. Formally, for every $m, m' \in \mathcal{M}$, and every $c \in \mathcal{C}$,

$$\Pr[\mathsf{Enc}_K(m) = c] = \Pr[\mathsf{Enc}_K(m') = c] \tag{2.1}$$

(where the probabilities are over choice of K and any randomness of Enc). This implies that the ciphertext contains no information about the plaintext, and that it is impossible to distinguish an encryption of m from an encryption of m', since the distributions over the ciphertext are the same in each case.

LEMMA 2.4 *An encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with message space* $\mathcal{M}$ *is perfectly secret if and only if Equation (2.1) holds for every* $m, m' \in \mathcal{M}$ *and every* $c \in \mathcal{C}$.

PROOF We show that if the stated condition holds, then the scheme is perfectly secret; the converse implication is left to Exercise 2.4. Fix a distribution over $\mathcal{M}$, a message m, and a ciphertext c for which $\Pr[C = c] > 0$. If $\Pr[M = m] = 0$ then we trivially have

$$\Pr[M = m \mid C = c] = 0 = \Pr[M = m].$$

So, assume $\Pr[M = m] > 0$. Notice first that

$$\Pr[C = c \mid M = m] = \Pr[\mathsf{Enc}_K(M) = c \mid M = m] = \Pr[\mathsf{Enc}_K(m) = c],$$

where the first equality is by definition of the random variable C, and the second is because we condition on the event that M is equal to m. Set $\delta_c \stackrel{\text{def}}{=} \Pr[\mathsf{Enc}_K(m) = c] = \Pr[C = c \mid M = m]$. If the condition of the lemma holds, then for *every* $m' \in \mathcal{M}$ we have $\Pr[\mathsf{Enc}_K(m') = c] = \Pr[C = c \mid M = m'] = \delta_c$. Using Bayes' Theorem (see Appendix A.3), we thus have

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

where the summation is over $m' \in \mathcal{M}$ with $\Pr[M = m'] \neq 0$. We conclude that for every $m \in \mathcal{M}$ and $c \in \mathcal{C}$ for which $\Pr[C = c] > 0$, it holds that $\Pr[M = m \mid C = c] = \Pr[M = m]$, and so the scheme is perfectly secret. ■

Perfect (adversarial) indistinguishability. We conclude this section by presenting another equivalent definition of perfect secrecy. This definition is based on an *experiment* involving an adversary passively observing a ciphertext and then trying to guess which of two possible messages was encrypted. We introduce this notion since it will serve as our starting point for defining computational security in the next chapter. Indeed, throughout the rest of the book we will often use experiments of this sort to define security.

In the present context, we consider the following experiment: an adversary $\mathcal{A}$ first specifies two arbitrary messages $m_0, m_1 \in \mathcal{M}$. One of these two

messages is chosen uniformly at random and encrypted using a random key; the resulting ciphertext is given to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a "guess" as to which of the two messages was encrypted; $\mathcal{A}$ *succeeds* if it guesses correctly. An encryption scheme is *perfectly indistinguishable* if *no* adversary $\mathcal{A}$ can succeed with probability better than 1/2. (Note that, for any encryption scheme, $\mathcal{A}$ can succeed with probability 1/2 by outputting a uniform guess; the requirement is simply that no attacker can do any better than this.) We stress that no limitations are placed on the computational power of $\mathcal{A}$.

Formally, let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme with message space $\mathcal{M}$. Let $\mathcal{A}$ be an adversary, which is formally just a (stateful) algorithm. We define an experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$ 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 key k is generated using* Gen*, and a uniform bit $b \in \{0,1\}$ is chosen. Ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$. We refer to c as the* challenge ciphertext*.*
3. *$\mathcal{A}$ outputs a bit b'.*
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. We write $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1$ if the output of the experiment is 1 and in this case we say that $\mathcal{A}$* succeeds*.*

As noted earlier, it is trivial for $\mathcal{A}$ to succeed with probability 1/2 by outputting a random guess. Perfect indistinguishability requires that it is impossible for any $\mathcal{A}$ to do better.

DEFINITION 2.5 *Encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with message space $\mathcal{M}$ is* perfectly indistinguishable *if for every $\mathcal{A}$ it holds that*

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

The following lemma states that Definition 2.5 is equivalent to Definition 2.3. We leave the proof of the lemma as Exercise 2.5.

LEMMA 2.6 *Encryption scheme Π is perfectly secret if and only if it is perfectly indistinguishable.*

Example 2.7
We show that the Vigenère cipher is *not* perfectly indistinguishable, at least for certain parameters. Concretely, let Π denote the Vigenère cipher for the message space of two-character strings, and where the period is chosen uniformly in $\{1, 2\}$. To show that Π is not perfectly indistinguishable, we exhibit an adversary $\mathcal{A}$ for which $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1\right] > \frac{1}{2}$.

Adversary $\mathcal{A}$ does:

1. Output $m_0 = \mathtt{aa}$ and $m_1 = \mathtt{ab}$.
2. Upon receiving the challenge ciphertext $c = c_1c_2$, do the following: if $c_1 = c_2$ output 0; else output 1.

Computation of $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1\right]$ is tedious but straightforward.

$$\begin{aligned}
&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1\right] \\
&= \frac{1}{2} \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1 \mid b = 0\right] + \frac{1}{2} \cdot \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1 \mid b = 1\right] \\
&= \frac{1}{2} \cdot \Pr[\mathcal{A} \text{ outputs } 0 \mid b = 0] + \frac{1}{2} \cdot \Pr[\mathcal{A} \text{ outputs } 1 \mid b = 1], \qquad (2.2)
\end{aligned}$$

where b is the uniform bit determining which message gets encrypted. $\mathcal{A}$ outputs 0 if and only if the two characters of the ciphertext $c = c_1c_2$ are equal. When $b = 0$ (so $m_0 = \mathtt{aa}$ is encrypted) then $c_1 = c_2$ if either (1) a key of period 1 is chosen, or (2) a key of period 2 is chosen, and both characters of the key are equal. The former occurs with probability $\frac{1}{2}$, and the latter occurs with probability $\frac{1}{2} \cdot \frac{1}{26}$. So

$$\Pr[\mathcal{A} \text{ outputs } 0 \mid b = 0] = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{26} \approx 0.52.$$

When $b = 1$ then $c_1 = c_2$ only if a key of period 2 is chosen and the first character of the key is one more than the second character of the key, which happens with probability $\frac{1}{2} \cdot \frac{1}{26}$. So

$$\Pr[\mathcal{A} \text{ outputs } 1 \mid b = 1] = 1 - \Pr[\mathcal{A} \text{ outputs } 0 \mid b = 1] = 1 - \frac{1}{2} \cdot \frac{1}{26} \approx 0.98.$$

Plugging into Equation (2.2) then gives

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1\right] = \frac{1}{2} \cdot \left(\frac{1}{2} + \frac{1}{2} \cdot \frac{1}{26} + 1 - \frac{1}{2} \cdot \frac{1}{26}\right) = 0.75 > \frac{1}{2},$$

and the scheme is not perfectly indistinguishable. ♢

2.2 The One-Time Pad

In 1917, Vernam patented a perfectly secret encryption scheme now called the *one-time pad*. At the time Vernam proposed the scheme, there was no proof that it was perfectly secret; in fact, there was not yet a notion of what perfect secrecy was. Approximately 25 years later, however, Shannon introduced the definition of perfect secrecy and demonstrated that the one-time pad achieves that level of security.

CONSTRUCTION 2.8

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

- Gen: the key-generation algorithm chooses a key 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 $2^{-\ell}$).
- Enc: given a key $k \in \{0,1\}^\ell$ and a message $m \in \{0,1\}^\ell$, the encryption algorithm outputs the ciphertext $c := k \oplus m$.
- Dec: given a key $k \in \{0,1\}^\ell$ and a ciphertext $c \in \{0,1\}^\ell$, the decryption algorithm outputs the message $m := k \oplus c$.

The one-time pad encryption scheme.

In describing the scheme we let $a \oplus b$ denote the *bitwise exclusive-or* (XOR) of two binary strings a and b (i.e., if $a = a_1 \cdots a_\ell$ and $b = b_1 \cdots b_\ell$ are ℓ-bit strings, then $a \oplus b$ is the ℓ-bit string given by $a_1 \oplus b_1 \cdots a_\ell \oplus b_\ell$). In the one-time pad encryption scheme the key is a uniform string of the same length as the message; the ciphertext is computed by simply XORing the key and the message. A formal definition is given as Construction 2.8. Before discussing security, we first verify correctness: for every key k and every message 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 valid encryption scheme.

One can easily prove perfect secrecy of the one-time pad using Lemma 2.4 and the fact that the ciphertext is uniformly distributed regardless of what message is encrypted. We give a proof based directly on the original definition.

THEOREM 2.9 *The one-time pad encryption scheme is perfectly secret.*

PROOF We first compute $\Pr[C = c \mid M = m']$ for arbitrary $c \in \mathcal{C}$ and $m' \in \mathcal{M}$. For the one-time pad,

$$\begin{aligned}\Pr[C = c \mid M = m'] = \Pr[\mathsf{Enc}_K(m') = c] &= \Pr[m' \oplus K = c] \\ &= \Pr[K = m' \oplus c] \\ &= 2^{-\ell},\end{aligned}$$

where the final equality holds because the key K is a uniform ℓ-bit string. Fix any distribution over $\mathcal{M}$. For any $c \in \mathcal{C}$, we have

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

where the sum is over $m' \in \mathcal{M}$ with $\Pr[M = m'] \neq 0$. Bayes' Theorem gives:

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

We conclude that the one-time pad is perfectly secret. ■

The one-time pad was used by several national-intelligence agencies in the mid-20th century to encrypt sensitive traffic. Perhaps most famously, the "red phone" linking the White House and the Kremlin during the Cold War was protected using one-time pad encryption, where the governments of the US and USSR would exchange extremely long keys using trusted couriers carrying briefcases of paper on which random characters were written.

Notwithstanding the above, one-time pad encryption is rarely used any more due to a number of drawbacks it has. Most prominent is that *the key is as long as the message*.[2] This limits the usefulness of the scheme for sending very long messages (as it may be difficult to securely share and store a very long key), and is problematic when the parties cannot predict in advance (an upper bound on) how long the message will be.

Moreover, the one-time pad—as the name indicates—*is only secure if used once* (with the same key). Although we did not yet define a notion of secrecy when multiple messages are encrypted, it is easy to see that encrypting more than one message with the same key 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

$$c \oplus c' = (m \oplus k) \oplus (m' \oplus k) = m \oplus m'$$

and thus learn the exclusive-or of the two messages or, equivalently, exactly where the two messages differ. While this may not seem very significant, it is enough to rule out any claims of perfect secrecy for encrypting two messages using the same key. Moreover, if the messages correspond to natural-language text, then given the exclusive-or of two sufficiently long messages it is possible to perform frequency analysis (as in the previous chapter, though more complex) and recover the messages themselves. An interesting historical example of this is given by the *VENONA project*, as part of which the US and UK were able to decrypt ciphertexts sent by the Soviet Union that were mistakenly encrypted with repeated portions of a one-time pad over several decades.

[2] This does not make the one-time pad useless, since it may be easier for two parties to share a key at some point in time before the message to be communicated is known.

2.3 Limitations of Perfect Secrecy

We ended the previous section by noting some drawbacks of the one-time pad encryption scheme. Here, we show that these drawbacks are not specific to that scheme, but are instead *inherent* limitations of perfect secrecy. 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 all keys are the same length, and the message space consists of all strings of some fixed length, this implies that the key is at least as long as the message. In particular, the key length of the one-time pad is optimal. (The other limitation—namely, that the key can be used only once—is also inherent if perfect secrecy is required; see Exercise 2.13.)

THEOREM 2.10 *If* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a perfectly secret encryption scheme with message space* $\mathcal{M}$ *and key space* $\mathcal{K}$*, then* $|\mathcal{K}| \geq |\mathcal{M}|$*.*

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

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

Clearly $|\mathcal{M}(c)| \leq |\mathcal{K}|$. (Recall that we may assume Dec is deterministic.) If $|\mathcal{K}| < |\mathcal{M}|$, there is some $m' \in \mathcal{M}$ such that $m' \notin \mathcal{M}(c)$. But then

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

and so the scheme is not perfectly secret. ■

Perfect secrecy with shorter keys? The above theorem shows an inherent limitation of schemes that achieve perfect secrecy. Even so, individuals occasionally claim they have developed a radically new encryption scheme that is "unbreakable" and achieves the security of the one-time pad without using keys as long as what is being encrypted. The above proof demonstrates that such claims *cannot* be true; anyone making such claims either knows very little about cryptography or is blatantly lying.

2.4 *Shannon's Theorem

In his work on perfect secrecy, Shannon also provided a characterization of perfectly secret encryption schemes. This characterization says that, under certain conditions, the key-generation algorithm Gen must choose the key *uniformly* from the set of all possible keys (as in the one-time pad); moreover, for every message m and ciphertext c there is a *unique* key mapping m to c (again, as in the one-time pad). Beyond being interesting in its own right, this theorem is a useful tool for proving (or disproving) perfect secrecy of suggested schemes. We discuss this further after the proof.

The theorem as stated here assumes $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$, meaning that the sets of plaintexts, keys, and ciphertexts all have the same size. We have already seen that for perfect secrecy we must have $|\mathcal{K}| \geq |\mathcal{M}|$. It is easy to see that correct decryption requires $|\mathcal{C}| \geq |\mathcal{M}|$. Therefore, in some sense, encryption schemes with $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$ are "optimal."

THEOREM 2.11 (Shannon's theorem) *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be an encryption scheme with message space* $\mathcal{M}$*, for which* $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$*. The scheme is perfectly secret if and only if:*

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

PROOF The intuition behind the proof is as follows. To see that the stated conditions imply perfect secrecy, note that condition 2 means that any ciphertext c could be the result of encrypting any possible plaintext m, because there is some key k mapping m to c. Since there is a *unique* such key, and each key is chosen with equal probability, perfect secrecy follows as for the one-time pad. For the other direction, perfect secrecy immediately implies that for every m and c there is at least one key mapping m to c. The fact that $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$ means, moreover, that for every m and c there is exactly *one* such key. Given this, each key must be chosen with equal probability or else perfect secrecy would fail to hold. A formal proof follows.

We assume for simplicity that Enc is deterministic. (One can show that this is without loss of generality here.) We first prove that if the encryption scheme satisfies conditions 1 and 2, then it is perfectly secret. The proof is essentially the same as the proof of perfect secrecy for the one-time pad, so we will be relatively brief. Fix arbitrary $c \in \mathcal{C}$ and $m \in \mathcal{M}$. Let k be the unique key, guaranteed by condition 2, for which $\mathsf{Enc}_k(m) = c$. Then,

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

where the final equality holds by condition 1. So

$$\Pr[C = c] = \sum_{m \in \mathcal{M}} \Pr[\mathsf{Enc}_K(m) = c] \cdot \Pr[M = m] = 1/|\mathcal{K}|.$$

This holds for any distribution over $\mathcal{M}$. Thus, for any distribution over $\mathcal{M}$, any $m \in \mathcal{M}$ with $\Pr[M = m] \neq 0$, and any $c \in \mathcal{C}$, we have:

$$\begin{aligned}\Pr[M = m \mid C = c] &= \frac{\Pr[C = c \mid M = m] \cdot \Pr[M = m]}{\Pr[C = c]} \\ &= \frac{\Pr[\mathsf{Enc}_K(m) = c] \cdot \Pr[M = m]}{\Pr[C = c]} \\ &= \frac{|\mathcal{K}|^{-1} \cdot \Pr[M = m]}{|\mathcal{K}|^{-1}} \;=\; \Pr[M = m],\end{aligned}$$

and the scheme is perfectly secret.

For the second direction, assume the encryption scheme is perfectly secret; we show that conditions 1 and 2 hold. Fix arbitrary $c \in \mathcal{C}$. There must be some message m^* for which $\Pr[\mathsf{Enc}_K(m^*) = c] \neq 0$. Lemma 2.4 then implies that $\Pr[\mathsf{Enc}_K(m) = c] \neq 0$ for every $m \in \mathcal{M}$. In other words, if we let $\mathcal{M} = \{m_1, m_2, \ldots\}$, then for each $m_i \in \mathcal{M}$ we have a nonempty set of keys $\mathcal{K}_i \subset \mathcal{K}$ such that $\mathsf{Enc}_k(m_i) = c$ if and only if $k \in \mathcal{K}_i$. Moreover, when $i \neq j$ then $\mathcal{K}_i$ and $\mathcal{K}_j$ must be disjoint or else correctness fails to hold. Since $|\mathcal{K}| = |\mathcal{M}|$, we see that each $\mathcal{K}_i$ contains only a single key k_i, as required by condition 2. Now, Lemma 2.4 shows that for any $m_i, m_j \in \mathcal{M}$ we have

$$\Pr[K = k_i] = \Pr[\mathsf{Enc}_K(m_i) = c] = \Pr[\mathsf{Enc}_K(m_j) = c] = \Pr[K = k_j].$$

Since this holds for all $1 \leq i, j \leq |\mathcal{M}| = |\mathcal{K}|$, and $k_i \neq k_j$ for $i \neq j$, this means each key is chosen with probability $1/|\mathcal{K}|$, as required by condition 1. ■

Shannon's theorem is useful for deciding whether a given scheme is perfectly secret. Condition 1 is easy to check, and condition 2 can be demonstrated (or contradicted) without having to compute any probabilities (in contrast to working with Definition 2.3 directly). As an example, perfect secrecy of the one-time pad is trivial to prove using Shannon's theorem. We stress, however, that the theorem only applies when $|\mathcal{M}| = |\mathcal{K}| = |\mathcal{C}|$.

References and Additional Reading

The one-time pad is popularly credited to Vernam [172], who filed a patent on it, but recent historical research [25] shows that it was invented some

35 years earlier. Analysis of the one-time pad had to await the groundbreaking work of Shannon [154], who introduced the notion of perfect secrecy.

In this chapter we studied perfectly secret *encryption*. Some other cryptographic problems 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 (undetectably) modifying a message sent from one party to another. We study this problem in depth in Chapter 4, discussing "perfectly secure" message authentication in Section 4.6.

Exercises

2.1 Prove that, by redefining the key space, we may assume that the key-generation algorithm Gen chooses a key uniformly at random from the key space, without changing $\Pr[C = c \mid M = m]$ for any m, c.

> **Hint:** Define the key space to be the set of all possible random tapes for the randomized algorithm Gen.

2.2 Prove that, by redefining the key space, we may assume that Enc is deterministic without changing $\Pr[C = c \mid M = m]$ for any m, c.

2.3 Prove or refute: An encryption scheme with message space $\mathcal{M}$ is perfectly secret if and only if for every probability distribution over $\mathcal{M}$ and every $c_0, c_1 \in \mathcal{C}$ we have $\Pr[C = c_0] = \Pr[C = c_1]$.

2.4 Prove the second direction of Lemma 2.4.

2.5 Prove Lemma 2.6.

2.6 For each of the following encryption schemes, state whether the scheme is perfectly secret. Justify your answer in each case.

(a) The message space is $\mathcal{M} = \{0, \ldots, 4\}$. Algorithm Gen chooses a uniform key from the key space $\{0, \ldots, 5\}$. $\mathsf{Enc}_k(m)$ returns $[k + m \bmod 5]$, and $\mathsf{Dec}_k(c)$ returns $[c - k \bmod 5]$.

(b) The message space is $\mathcal{M} = \{m \in \{0,1\}^\ell \mid \text{the last bit of } m \text{ is } 0\}$. Gen chooses a uniform key from $\{0,1\}^{\ell-1}$. $\mathsf{Enc}_k(m)$ returns ciphertext $m \oplus (k \| 0)$, and $\mathsf{Dec}_k(c)$ returns $c \oplus (k \| 0)$.

2.7 When using the one-time pad with the key $k = 0^\ell$, we have $\mathsf{Enc}_k(m) = k \oplus m = m$ and the message is sent in the clear! It has therefore been suggested to modify the one-time pad by only encrypting with $k \neq 0^\ell$ (i.e., to have Gen choose k uniformly from the set of *nonzero* keys of length ℓ). Is this modified scheme still perfectly secret? Explain.

2.8 Let Π denote the Vigenère cipher where the message space consists of all 3-character strings (over the English alphabet), and the key is generated by first choosing the period t uniformly from $\{1, 2, 3\}$ and then letting the key be a uniform string of length t.

(a) Define $\mathcal{A}$ as follows: $\mathcal{A}$ outputs $m_0 = \texttt{aab}$ and $m_1 = \texttt{abb}$. When given a ciphertext c, it outputs 0 if the first character of c is the same as the second character of c, and outputs 1 otherwise. Compute $\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1]$.

(b) Construct and analyze an adversary $\mathcal{A}'$ for which $\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi} = 1]$ is greater than your answer from part (a).

2.9 In this exercise, we look at different conditions under which the shift, mono-alphabetic substitution, and Vigenère ciphers are perfectly secret:

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

(b) What is the largest message space $\mathcal{M}$ for which the mono-alphabetic substitution cipher provides perfect secrecy?

(c) Prove that the Vigenère cipher using (fixed) period t is perfectly secret when used to encrypt messages of length t.

Reconcile this with the attacks shown in the previous chapter.

2.10 Prove that a scheme satisfying Definition 2.5 must have $|\mathcal{K}| \geq |\mathcal{M}|$ without using Lemma 2.4. Specifically, let Π be an arbitrary encryption scheme with $|\mathcal{K}| < |\mathcal{M}|$. Show an $\mathcal{A}$ for which $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1\right] > \frac{1}{2}$.

Hint: It may be easier to let $\mathcal{A}$ be randomized.

2.11 Assume we require only that an encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with message space $\mathcal{M}$ satisfy the following: For all $m \in \mathcal{M}$, we have $\Pr[\mathsf{Dec}_K(\mathsf{Enc}_K(m)) = m] \geq 2^{-t}$. (This probability is taken over choice of the key as well as any randomness used during encryption.) Show that perfect secrecy can be achieved with $|\mathcal{K}| < |\mathcal{M}|$ when $t \geq 1$. Prove a lower bound on the size of $\mathcal{K}$ in terms of t.

2.12 Let $\varepsilon \geq 0$ be a constant. Say an encryption scheme is *ε-perfectly secret* if for every adversary $\mathcal{A}$ it holds that

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

(Compare to Definition 2.5.) Show that ε-perfect secrecy can be achieved with $|\mathcal{K}| < |\mathcal{M}|$ when $\varepsilon > 0$. Prove a lower bound on the size of $\mathcal{K}$ in terms of ε.

2.13 In this problem we consider definitions of perfect secrecy for the encryption of *two* messages (using the same key). Here we consider distributions over *pairs* of messages from the message space $\mathcal{M}$; we let M_1, M_2 be random variables denoting the first and second message, respectively. (We stress that these random variables are not assumed to be independent.) We generate a (single) key k, sample a pair of messages (m_1, m_2) according to the given distribution, and then compute ciphertexts $c_1 \leftarrow \mathsf{Enc}_k(m_1)$ and $c_2 \leftarrow \mathsf{Enc}_k(m_2)$; this induces a distribution over pairs of ciphertexts and we let C_1, C_2 be the corresponding random variables.

(a) Say encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *perfectly secret for two messages* if for all distributions over $\mathcal{M} \times \mathcal{M}$, all $m_1, m_2 \in \mathcal{M}$, and all ciphertexts $c_1, c_2 \in \mathcal{C}$ with $\Pr[C_1 = c_1 \wedge C_2 = c_2] > 0$:

$$\Pr[M_1 = m_1 \wedge M_2 = m_2 \mid C_1 = c_1 \wedge C_2 = c_2] = \Pr[M_1 = m_1 \wedge M_2 = m_2].$$

Prove that *no* encryption scheme can satisfy this definition.

Hint: Take $c_1 = c_2$.

(b) Say encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *perfectly secret for two distinct messages* if for all distributions over $\mathcal{M} \times \mathcal{M}$ where the first and second messages are guaranteed to be different (i.e., distributions over pairs of *distinct* messages), all $m_1, m_2 \in \mathcal{M}$, and all $c_1, c_2 \in \mathcal{C}$ with $\Pr[C_1 = c_1 \wedge C_2 = c_2] > 0$:

$$\Pr[M_1 = m_1 \wedge M_2 = m_2 \mid C_1 = c_1 \wedge C_2 = c_2] = \Pr[M_1 = m_1 \wedge M_2 = m_2].$$

Show an encryption scheme that provably satisfies this definition.

Hint: The encryption scheme you propose need not be efficient, although an efficient solution is possible.

Part II

Private-Key (Symmetric) Cryptography

Chapter 3

Private-Key Encryption

In the previous chapter we saw some fundamental limitations of perfect secrecy. In this chapter we begin our study of modern cryptography by introducing the weaker (but sufficient) notion of *computational* secrecy. We will then show how this definition can be used to bypass the impossibility results shown previously and, in particular, how a short key (say, 128 bits long) can be used to encrypt many long messages (say, gigabytes in total).

Along the way we will study the fundamental notion of *pseudorandomness*, which captures the idea that something can "look" completely random even though it is not. This powerful concept underlies much of modern cryptography, and has applications and implications beyond the field as well.

3.1 Computational Security

In Chapter 2 we introduced the notion of perfect secrecy. While perfect secrecy is a worthwhile goal, it is also unnecessarily strong. Perfect secrecy requires that *absolutely no information* about an encrypted message is leaked, even to an eavesdropper *with unlimited computational power*. For all practical purposes, however, an encryption scheme would still be considered secure if it leaked only a *tiny* amount of information to eavesdroppers with *bounded computational power*. For example, a scheme that leaks information with probability at most 2^{-60} to eavesdroppers investing up to 200 years of computational effort on the fastest available supercomputer is adequate for any real-world application. Security definitions that take into account computational limits on the attacker, and allow for a small probability of failure, are called *computational*, to distinguish them from notions (like perfect secrecy) that are *information-theoretic* in nature. Computational security is now the *de facto* way in which security is defined for all cryptographic purposes.

We stress that although we give up on obtaining perfect security, this does not mean we do away with the rigorous mathematical approach. Definitions and proofs are still essential, and the only difference is that we now consider weaker (but still meaningful) definitions of security.

Computational security incorporates two relaxations relative to information-

theoretic notions of security (in the case of encryption, both these relaxations are necessary in order to go beyond the limitations of perfect secrecy discussed in the previous chapter):

1. *Security is only guaranteed against* efficient *adversaries that run for some feasible amount of time.* This means that given enough time (or sufficient computational resources) an attacker may be able to violate security. If we can make the resources required to break the scheme larger than those available to any realistic attacker, then for all practical purposes the scheme is unbreakable.

2. *Adversaries can potentially succeed (i.e., security can potentially fail) with some* very small probability. If we can make this probability sufficiently small, we need not worry about it.

To obtain a meaningful theory, we need to precisely define the above relaxations. There are two general approaches for doing so: the *concrete approach* and the *asymptotic approach.* These are described next.

3.1.1 The Concrete Approach

The concrete approach to computational security quantifies the security of a cryptographic scheme by explicitly bounding the maximum success probability of any (randomized) adversary running for some specified amount of time or, more precisely, investing some specific amount of computational effort. Thus, a concrete definition of security takes roughly the following form:

> *A scheme is* (t, ε)-secure *if any 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 in question.) As an example, one might have 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 2^{-60}. Or, it may be more convenient to measure running time in terms of CPU cycles, and to construct a scheme such that no adversary using at most 2^{80} cycles can break the scheme with probability better than 2^{-60}.

It is instructive to get a feel for the large values of t and the small values of 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—and so the key space has size 2^n—an adversary running for time t (measured in, say, computer cycles) succeeds in breaking the scheme with probability at most

$ct/2^n$ for some fixed constant c. (This simply corresponds to a brute-force search of the key space, and assumes no preprocessing has been done.)

Assuming $c = 1$ for simplicity, a key of length $n = 60$ provides adequate security against an adversary using a desktop computer. Indeed, on a 4 GHz processor (that executes 4×10^9 cycles per second) 2^{60} CPU cycles require $2^{60}/(4 \times 10^9)$ seconds, or about 9 years. However, the fastest supercomputer at the time of this writing can execute roughly 2×10^{16} floating point operations per second, and 2^{60} such operations require only about 1 minute on such a machine. Taking $n = 80$ would be a more prudent choice; even the computer just mentioned would take about 2 years to carry out 2^{80} operations.

(The above numbers are for illustrative purposes only; in practice $c > 1$, and several other factors—such as the time required for memory access and the possibility of parallel computation on a network of computers—significantly affect the performance of brute-force attacks.)

Today, however, a recommended key length might be $n = 128$. The difference between 2^{80} and 2^{128} is a *multiplicative factor* of 2^{48}. 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}.

If the probability that an attacker can successfully recover an encrypted message in one year is at most 2^{-60}, then it is much more likely that the sender and receiver will both be hit by lightning in that same period of time. 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 is important in practice, since concrete guarantees are what users of a cryptographic scheme are ultimately interested in. However, precise concrete guarantees are difficult to provide. Furthermore, one must be careful in interpreting concrete security claims. For example, a claim that no adversary running for 5 years can break a given scheme with probability better than ε begs the questions: what type of computing power (e.g., desktop PC, supercomputer, network of hundreds of computers) does this assume? Does this take into account future advances in computing power (which, by Moore's Law, roughly doubles every 18 months)? Does the estimate assume the use of "off-the-shelf" algorithms, or dedicated software implementations 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.

3.1.2 The Asymptotic Approach

As partly noted above, there are some technical and theoretical difficulties in using the concrete-security approach. These issues must be dealt with in

practice, but when concrete security is not an immediate concern it is convenient instead to use an *asymptotic* approach to security; this is the approach taken in this book. This approach, rooted in complexity theory, introduces an integer-valued *security parameter* (denoted by n) that parameterizes both cryptographic schemes as well as all involved parties (namely, the honest parties as well as the attacker). When honest parties initialize a scheme (i.e., when they generate keys), they choose some value n for the security parameter; for the purposes of this discussion, one can think of the security parameter as corresponding to the length of the key. The security parameter is assumed to be known to any adversary attacking the scheme, and we now view the running time of the adversary, as well as its success probability, as functions of the security parameter rather than as concrete numbers. Then:

1. We equate "efficient adversaries" with randomized (i.e., probabilistic) algorithms running in time *polynomial in* n. This means there is some polynomial p such that the adversary runs for time at most $p(n)$ when the security parameter is n. We also require—for real-world efficiency—that honest parties run in polynomial time, although we stress that the adversary may be much more powerful (and run much longer than) the honest parties.

2. We equate the notion of "small probabilities of success" with success probabilities *smaller than any inverse polynomial in* n (see Definition 3.4). Such probabilities are called *negligible*.

Let PPT stand for "probabilistic polynomial-time." A definition of asymptotic security then takes the following general form:

> *A scheme is* secure *if any* PPT *adversary succeeds in breaking the scheme with at most negligible probability.*

This notion of security is *asymptotic* since security depends on the behavior of the scheme for sufficiently large values of n. The following example makes this clear.

Example 3.2
Say we have a scheme that is asymptotically 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 very useful. Even for $n = 50$ an adversary running for 50^3 minutes (about 3 months) can break the scheme with probability roughly $1/1000$, which may not be acceptable. On the other hand, when $n = 500$ an adversary running for 200 years breaks the scheme only with probability roughly 2^{-500}. $\diamondsuit$

As indicated by the previous example, we can view the security parameter as a mechanism that allows the honest parties to "tune" the security of a scheme to some desired level. (Increasing the security parameter also increases the time required to run the scheme, as well as the length of the key, so the honest parties will want to set the security parameter as small as possible subject to defending against the class of attacks they are concerned about.) Viewing the security parameter as the key length, this corresponds roughly to the fact that the time required for an exhaustive-search attack grows exponentially in the length of the key. The ability to "increase security" by increasing the security parameter has important practical ramifications, since it enables honest parties to defend against increases in computing power. The following example gives a sense of how this might play out in practice.

Example 3.3
Let us see the effect that the availability of faster computers might have on security in practice. Say we have a cryptographic scheme in which the honest parties 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 at most $2^{-n/2}$. (The numbers are intended to make calculations easier, and are not meant to correspond to any existing cryptographic scheme.)

Say all parties are using 2 GHz computers and the honest parties set $n = 80$. Then the honest parties run for $10^6 \cdot 6400$ cycles, or 3.2 seconds, and an adversary running for $10^8 \cdot (80)^4$ cycles, or roughly 3 weeks, can break the scheme with probability only 2^{-40}.

Say 8 GHz computers become available, and all parties upgrade. Honest parties can increase n to 160 (which requires generating a fresh key) and maintain a running time of 3.2 seconds (i.e., $10^6 \cdot 160^2$ cycles at $8 \cdot 10^9$ cycles/second). In contrast, the adversary now has to run for over 8 million seconds, or more than 13 weeks, to achieve a success probability of 2^{-80}. The effect of a faster computer has been to make the adversary's job *harder*. ♢

Even when using the asymptotic approach it is important to remember that, ultimately, when a cryptosystem is deployed in practice a concrete security guarantee will be needed. (After all, one must decide on some value of n.) As the above examples indicate, however, it is generally the case that an asymptotic security claim can be translated into a concrete security bound for any desired value of n.

The Asymptotic Approach in Detail

We now discuss more formally the notions of "polynomial-time algorithms" and "negligible success probabilities."

Efficient algorithms. We have defined an algorithm to be efficient if it runs in polynomial time. An algorithm A runs in polynomial time if there exists a

polynomial p 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.) As mentioned earlier, we are only interested in adversaries whose running time is polynomial in the security parameter n. Since we measure the running time of an algorithm in terms of the length of its input, we sometimes provide algorithms with the security parameter written in unary (i.e., as 1^n, or a string of n ones) as input. Parties (or, more precisely, the algorithms they run) may take other inputs besides the security parameter—for example, a message to be encrypted—and we allow their running time to be polynomial in the (total) length of their inputs.

By default, we allow all algorithms to be probabilistic (or randomized). Any such algorithm may "toss a coin" at each step of its execution; this is a metaphorical way of saying that the algorithm can access an unbiased random bit at each step. Equivalently, we can view a randomized algorithm as one that, in addition to its input, is given a uniformly distributed *random tape* of sufficient length[1] whose bits it can use, as needed, throughout its execution.

We consider randomized algorithms by default for two reasons. 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 this, it is natural to allow adversaries to be probabilistic as well. Second, randomization is practical and—as far as we know—gives attackers additional power. Since our goal is to model *all* realistic attacks, we prefer a more liberal definition of efficient computation.

Negligible success probability. A negligible function is one that is asymptotically smaller than any inverse polynomial function. Formally:

DEFINITION 3.4 *A function f from the natural numbers to the non-negative real numbers is* negligible *if for every positive polynomial p there is an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

For shorthand, the above is also stated as follows: for every polynomial p and *all sufficiently large values of* 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 $f(n) < n^{-c}$. We typically denote an arbitrary negligible function by negl.

Example 3.5
The functions $2^{-n}, 2^{-\sqrt{n}}$, and $n^{-\log n}$ are all negligible. However, they approach zero at very different rates. For example, we can look at the minimum value of n for which each function is smaller than $1/n^5$:

[1] If the algorithm in question runs for $p(n)$ steps on inputs of length n, then a random tape of length $p(n)$ is sufficient since the attacker can read at most one random bit per time step.

1. Solving $2^{-n} < n^{-5}$ we get $n > 5 \log n$. The smallest integer value of n for which this holds is $n = 23$.
2. Solving $2^{-\sqrt{n}} < n^{-5}$ we get $n > 25 \log^2 n$. The smallest integer value of n for which this holds is $n \approx 3500$.
3. Solving $n^{-\log n} < n^{-5}$ we get $\log n > 5$. The smallest integer value of n for which this holds is $n = 33$.

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

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

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

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

The second part of the above proposition implies that if a certain event occurs with only negligible probability in a certain experiment, then the event occurs with negligible probability even if the experiment is repeated polynomially many times. (This relies on the union bound; see Proposition A.7.) For example, the probability that n fair coin flips all come up "heads" is negligible. This means that even if we repeat the experiment of flipping n coins polynomially many times, the probability that *any* of those experiments result in n heads is still negligible.

A corollary of the second part of the above proposition is that if a function g is *not* negligible, then neither is the function $f(n) \stackrel{\text{def}}{=} g(n)/p(n)$ for any positive polynomial p.

Asymptotic Security: A Summary

Any security definition consists of two parts: a definition of what is considered a "break" of the scheme, and a specification of the power of the adversary. The power of the adversary can relate to many issues (e.g., in the case of encryption, whether we assume a ciphertext-only attack or a chosen-plaintext attack). However, when it comes to the *computational* power of the adversary, we will from now on model the adversary as efficient and thus only consider adversarial strategies that can be implemented in probabilistic polynomial

time. Definitions will also always be formulated so that a break that occurs with negligible probability is not considered significant. Thus, the general framework of any security definition will be as follows:

> A scheme is *secure* if for every *probabilistic polynomial-time* adversary $\mathcal{A}$ carrying out an attack of some formally specified type, the probability that $\mathcal{A}$ succeeds in the attack (where success is also formally specified) 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 expand the term "negligible" in the above statement:

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

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

On the Choices Made in Defining Asymptotic Security

In defining the general notion of asymptotic security, we have made two choices: we have identified efficient adversarial strategies with the class of *probabilistic, polynomial-time algorithms*, and have equated small chances of success with *negligible probabilities*. Both of these choices are—to some extent—arbitrary, and one could build a perfectly reasonable theory by defining, say, efficient strategies as those running in quadratic time, or small success probabilities as those bounded by 2^{-n}. Nevertheless, we briefly justify the choices we have made (which are the standard ones).

Those familiar with complexity theory or algorithms will recognize that the idea of equating efficient computation with (probabilistic) polynomial-time algorithms is not unique to cryptography. One advantage of using (probabilistic) polynomial time as our measure of efficiency is that this frees us from having to precisely specify our model of computation, since the extended Church–Turing thesis states that all "reasonable" models of computation are polynomially equivalent. Thus, we need not specify whether we use Turing machines, boolean circuits, or random-access machines; we can present algorithms in high-level pseudocode and be confident that if our analysis shows that these algorithms run in polynomial time, then any reasonable implementation will also.

Another advantage of (probabilistic) polynomial-time algorithms is that they satisfy desirable closure properties: in particular, an algorithm that makes polynomially many calls to a polynomial-time subroutine (and does only polynomial computation in addition) will itself run in polynomial time.

The most important feature of negligible probabilities is the closure property we have already seen in Proposition 3.6(2): any polynomial times a negligible function is still negligible. This means, in particular, that if an algorithm makes polynomially many calls to some subroutine that "fails" with negligible probability each time it is called, then the probability that any of the calls to that subroutine fail is still negligible.

Necessity of the Relaxations

Computational secrecy introduces two relaxations of perfect secrecy: first, security is guaranteed only against efficient adversaries; second, a small probability of success is allowed. Both these relaxations are essential for achieving practical encryption schemes, and in particular for bypassing the negative results for perfectly secret encryption. We informally discuss why this is the case. Assume we have an encryption scheme where the size of the key space $\mathcal{K}$ is much smaller than the size of the message space $\mathcal{M}$. (As shown in the previous chapter, this means the scheme cannot be perfectly secret.) Two attacks apply regardless of how the encryption scheme is constructed:

- Given a ciphertext c, an adversary can decrypt c using all keys $k \in \mathcal{K}$. This gives a list of all the messages to which c can possibly correspond. Since this list cannot contain all of $\mathcal{M}$ (because $|\mathcal{K}| < |\mathcal{M}|$), this attack 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. Later, given a ciphertext c that is the encryption of an unknown message m, it is almost surely the case that $\mathsf{Dec}_k(c) = m$.

  Exhaustive-search attacks like the above allow an 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 messages $m_1, \ldots, m_\ell$. The adversary can *guess* a uniform key $k \in \mathcal{K}$ and check to see whether $\mathsf{Dec}_k(c_i) = m_i$ for all i. If so, then, as above, the attacker can use k to decrypt anything subsequently encrypted by the honest parties.

  Here the adversary runs in essentially constant time and succeeds with nonzero (though very small) probability $1/|\mathcal{K}|$.

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 the adversary does not have sufficient time to carry out a brute-force search) and are willing to allow a very small probability of success (so the second "attack" is ruled out).

3.2 Defining Computationally Secure Encryption

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

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

1. *The* key-generation algorithm Gen *takes as input* 1^n *(i.e., the security parameter written in unary) and outputs a key* k*; we write* $k \leftarrow \mathsf{Gen}(1^n)$ *(emphasizing that* Gen *is a randomized algorithm). We 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*. 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 *or an error. We assume that* Dec *is deterministic, and so write* $m := \mathsf{Dec}_k(c)$ *(assuming here that* Dec *does not return an error). We denote a generic error by the symbol* $\perp$*.*

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 for messages of length $\ell(n)$*.*

Almost always, $\mathsf{Gen}(1^n)$ simply outputs a uniform n-bit string as the key. When this is the case, we will omit Gen and simply define a private-key encryption scheme by a *pair* of algorithms $(\mathsf{Enc}, \mathsf{Dec})$.

The above definition considers *stateless* schemes, in which each invocation of Enc (and Dec) is independent of all prior invocations. Later in the chapter, we will occasionally discuss *stateful* schemes in which the sender (and possibly the receiver) is required to maintain state across invocations. Unless explicitly noted otherwise, all our results assume stateless encryption/decryption.

3.2.1 The Basic Definition of Security

We begin by presenting the most basic notion of security for private-key encryption: security against a ciphertext-only attack where the adversary observes only a *single* ciphertext or, equivalently, security when a given key is used to encrypt just a *single* message. We consider stronger definitions of security later in the chapter.

Motivating the definition. As we have already discussed, any definition of security consists of two distinct components: a threat model (i.e., a specification of the assumed power of the adversary) and a security goal (usually specified by describing what constitutes a "break" of the scheme). We begin our definitional treatment by considering the simplest threat model, where we have an *eavesdropping adversary* who observes the encryption of a single message. This is exactly the threat model that was considered in the previous chapter with the exception that, as explained in the previous section, we are now interested only in adversaries that are computationally bounded and so limited to running in polynomial time.

Although we have made two assumptions about the adversary's capabilities (namely, that it only eavesdrops, and that it runs in polynomial time), we make no assumptions whatsoever about the adversary's *strategy* in trying to decipher the ciphertext it observes. This is crucial for obtaining meaningful notions of security; the definition ensures protection against *any* computationally bounded adversary, regardless of the algorithm it uses.

Correctly defining the security goal for encryption is not trivial, but we have already discussed this issue at length in Section 1.4.1 and in the previous chapter. We therefore just recall that the idea behind the definition is that the adversary should be unable to learn *any partial information* about the plaintext from the ciphertext. The definition of *semantic security* (cf. Section 3.2.2) exactly formalizes this notion, and was the first definition of computationally secure encryption to be proposed. Semantic security is complex and difficult to work with. Fortunately, there is an equivalent definition called *indistinguishability* that is much simpler.

The definition of indistinguishability is patterned on the alternative definition of perfect secrecy given as Definition 2.5. (This serves as further motivation that the definition of indistinguishability is a good one.) Recall that Definition 2.5 considers an experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$ in which an adversary $\mathcal{A}$ outputs two messages m_0 and m_1, and is then given an encryption of one of those messages using a uniform key. The definition states that a scheme Π is secure if no adversary $\mathcal{A}$ can determine which of the messages m_0, m_1 was encrypted with probability any different from $1/2$, which is the probability that $\mathcal{A}$ is correct if it just makes a random guess.

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

1. We now consider only adversaries running in *polynomial time*, whereas Definition 2.5 considered even adversaries with unbounded running time.

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 other differences, the most prominent is that we now parameterize the experiment by a security parameter n. We then measure both the running time of the adversary $\mathcal{A}$ as well as its success probability as functions of n. We write $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ to denote the experiment being run with security parameter n, and write

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

to denote the probability that the output of experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$ is 1. Note that with $\mathcal{A}, \Pi$ fixed, Equation (3.1) is a function of n.

A second difference in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}$ is that we now explicitly require the adversary to output two messages m_0, m_1 *of equal length*. (In Definition 2.5 this requirement is implicit if the message space $\mathcal{M}$ only contains messages of some fixed length, as is the case for the one-time pad encryption scheme.) This means that, by default, we do not require a secure encryption scheme to hide the length of the plaintext. We revisit this point at the end of this section; see also Exercises 3.2 and 3.3.

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 with $|m_0| = |m_1|$.*
2. *A key k is generated by running $\mathsf{Gen}(1^n)$, and a uniform bit $b \in \{0,1\}$ is chosen. Ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$. We refer to c as the* challenge ciphertext.
3. *$\mathcal{A}$ outputs a bit b'.*
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise. If $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$, we say that $\mathcal{A}$* succeeds.

There is no limitation on the lengths of m_0 and m_1, as long as they are the same. (Of course, if $\mathcal{A}$ runs in polynomial time, then m_0 and m_1 have length polynomial in n.) If Π is a fixed-length scheme for messages of length $\ell(n)$, the above experiment is modified by requiring $m_0, m_1 \in \{0,1\}^{\ell(n)}$.

The fact that the adversary can only eavesdrop 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 makes the adversary significantly stronger.)

The definition of indistinguishability states that an encryption scheme is secure if no PPT adversary $\mathcal{A}$ succeeds in guessing which message was encrypted in the above experiment with probability significantly better than random guessing (which is correct with probability 1/2):

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

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

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

Note: unless otherwise qualified, when we write "$f(n) \leq g(n)$" we mean that inequality holds for all n.

It should be clear that Definition 3.8 is *weaker* than Definition 2.5, which is equivalent to perfect secrecy. Thus, any perfectly secret encryption scheme has indistinguishable encryptions in the presence of an eavesdropper. Our goal, therefore, will be to show that there exist encryption schemes satisfying the above in which the key is shorter than the message. That is, we will show schemes that satisfy Definition 3.8 but cannot satisfy Definition 2.5.

An equivalent formulation. Definition 3.8 requires that no PPT adversary can determine which of two messages was encrypted, with probability significantly better than 1/2. An equivalent formulation is that every PPT adversary *behaves the same* whether it sees an encryption of m_0 or of m_1. Since $\mathcal{A}$ outputs a single bit, "behaving the same" means it outputs 1 with almost the same probability in each case. To formalize this, define $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n, b)$ as above except that the fixed bit b is used (rather than being chosen at random). Let $\mathsf{out}_{\mathcal{A}}(\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n, b))$ denote the output bit b' of $\mathcal{A}$ in the experiment. The following essentially states that no $\mathcal{A}$ can determine whether it is running in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n, 0)$ or experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n, 1)$.

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

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

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

Encryption and Plaintext Length

The default notion of secure encryption does not require the encryption scheme to hide the plaintext length and, in fact, all commonly used encryption schemes reveal the plaintext length (or a close approximation thereof). The main reason for this is that it is *impossible* to support arbitrary-length messages while hiding all information about the plaintext length (cf. Exercise 3.2). In many cases this is inconsequential since the plaintext length is already public or is not sensitive. This is not always the case, however, and sometimes leaking the plaintext length is problematic. As examples:

- *Simple numeric/text data:* Say the encryption scheme being used reveals the plaintext length exactly. Then encrypted salary information would reveal whether someone makes a 5-figure or a 6-figure salary. Similarly, encryption of "yes"/"no" responses would leak the answer exactly.
- *Auto-suggestions:* Websites often include an "auto-complete" or "auto-suggestion" functionality by which the webserver suggests a list of potential words or phrases based on partial information the user has already typed. The *size* of this list can reveal information about the letters the user has typed so far. (For example, the number of auto-completions returned for "`th`" is far greater than the number for "`zo`.")
- *Database searches:* Consider a user querying a database for all records matching some search term. The *number of records returned* can reveal a lot of information about what the user was searching for. This can be particularly damaging if the user is searching for medical information and the query reveals information about a disease the user has.
- *Compressed data:* If the plaintext is compressed before being encrypted, then information about the plaintext might be revealed even if only fixed-length data is ever encrypted. (Such an encryption scheme would therefore not satisfy Definition 3.8.) For example, a short compressed plaintext would indicate that the original (uncompressed) plaintext has a lot of redundancy. If an adversary can control a portion of what gets encrypted, this vulnerability can enable an adversary to learn additional information about the plaintext; it has been shown possible to use an attack of exactly this sort (the *CRIME attack*) against encrypted HTTP traffic to reveal secret session cookies.

When using encryption one should determine whether leaking the plaintext length is a concern and, if so, take steps to mitigate or prevent such leakage by padding all messages to some pre-determined length before encrypting them.

3.2.2 *Semantic Security

We motivated the definition of secure encryption by saying that it should be infeasible for an adversary to learn any partial information about the plaintext

from the ciphertext. However, the definition of indistinguishability looks very different. As we have mentioned, Definition 3.8 is equivalent to a definition called *semantic security* that formalizes the notion that partial information cannot be learned. We build up to that definition by discussing two weaker notions and showing that they are implied by indistinguishability.

We begin by showing that indistinguishability means that ciphertexts leak no information about individual bits of the plaintext. Formally, say encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ is EAV-secure (recall then when Gen is omitted, the key is a uniform n-bit string), and $m \in \{0,1\}^\ell$ is uniform. Then we show that for any index i, it is infeasible to guess m^i from $\mathsf{Enc}_k(m)$ (where, in this section, m^i denotes the ith bit of m) with probability much better than 1/2.

THEOREM 3.10 *Let $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ be a fixed-length private-key encryption scheme for messages of length ℓ that has indistinguishable encryptions in the presence of an eavesdropper. Then for all* PPT *adversaries $\mathcal{A}$ and any $i \in \{1, \ldots, \ell\}$, there is 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 the probability is taken over uniform $m \in \{0,1\}^\ell$ and $k \in \{0,1\}^n$, the randomness of $\mathcal{A}$, and the randomness of Enc.

PROOF The idea behind the proof of this theorem is that if it were possible to guess the ith bit of m from $\mathsf{Enc}_k(m)$, then it would also be possible to distinguish between encryptions of messages m_0 and m_1 whose ith bits differ. We formalize this via a *proof by reduction*, in which we show how to use any efficient adversary $\mathcal{A}$ to construct an efficient adversary $\mathcal{A}'$ such that if $\mathcal{A}$ violates the security notion of the theorem for Π, then $\mathcal{A}'$ violates the definition of indistinguishability for Π. (See Section 3.3.2.) Since Π has indistinguishable encryptions, it must also be secure in the sense of the theorem.

Fix an arbitrary PPT adversary $\mathcal{A}$ and $i \in \{1, \ldots, \ell\}$. Let $I_0 \subset \{0,1\}^\ell$ be the set of all strings whose ith bit is 0, and let $I_1 \subset \{0,1\}^\ell$ be the set of all strings whose ith bit is 1. We have

$$\begin{aligned}&\Pr\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m)) = m^i\right] \\ &= \frac{1}{2} \cdot \Pr_{m_0 \leftarrow I_0}\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m_0)) = 0\right] + \frac{1}{2} \cdot \Pr_{m_1 \leftarrow I_1}\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m_1)) = 1\right].\end{aligned}$$

Construct the following eavesdropping adversary $\mathcal{A}'$:

Adversary $\mathcal{A}'$:

1. Choose uniform $m_0 \in I_0$ and $m_1 \in I_1$. Output m_0, m_1.
2. Upon observing a ciphertext c, invoke $\mathcal{A}(1^n, c)$. If $\mathcal{A}$ outputs 0, output $b' = 0$; otherwise, output $b' = 1$.

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

By the definition of experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n)$, we have that $\mathcal{A}'$ succeeds if and only if $\mathcal{A}$ outputs b upon receiving $\mathsf{Enc}_k(m_b)$. So

$$\begin{aligned}
&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1\right] \\
&= \Pr\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m_b)) = b\right] \\
&= \frac{1}{2} \cdot \Pr_{m_0 \leftarrow I_0}\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m_0)) = 0\right] + \frac{1}{2} \cdot \Pr_{m_1 \leftarrow I_1}\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m_1)) = 1\right] \\
&= \Pr\left[\mathcal{A}(1^n, \mathsf{Enc}_k(m)) = m^i\right].
\end{aligned}$$

By the assumption that $(\mathsf{Enc}, \mathsf{Dec})$ has indistinguishable encryptions in the presence of an eavesdropper, there is a negligible function negl such that $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)$. We conclude that

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

completing the proof. ■

We next claim, roughly, that indistinguishability means that no PPT adversary can learn *any* function of the plaintext given the ciphertext, regardless of the distribution of the message being sent. This is intended to capture the idea that no information about a plaintext is leaked by the resulting ciphertext. This requirement is, however, non-trivial to define formally. To see why, note that even for the case considered above, it is easy to compute the ith bit of m if m is chosen, say, uniformly from the set of all strings whose ith bit is 0 (rather than uniformly from $\{0,1\}^\ell$). Thus, what we actually want to say is that if there exists any adversary who correctly computes $f(m)$ with some probability when given $\mathsf{Enc}_k(m)$, then there exists an adversary that can correctly compute $f(m)$ with the same probability *without* being given the ciphertext at all (and only knowing the distribution of m). In what follows we focus on the case when m is chosen uniformly from some set $S \subseteq \{0,1\}^\ell$.

THEOREM 3.11 *Let* $(\mathsf{Enc}, \mathsf{Dec})$ *be a fixed-length private-key encryption scheme for messages of length* ℓ *that has indistinguishable encryptions in the presence of an eavesdropper. Then for any* PPT *algorithm* $\mathcal{A}$ *there is a* PPT *algorithm* $\mathcal{A}'$ *such that for any* $S \subseteq \{0,1\}^\ell$ *and any function* $f : \{0,1\}^\ell \to \{0,1\}$*, there is 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 the first probability is taken over uniform choice of $k \in \{0,1\}^n$ *and* $m \in S$*, the randomness of* $\mathcal{A}$*, and the randomness of* Enc*, and the second probability is taken over uniform choice of* $m \in S$ *and the randomness of* $\mathcal{A}'$*.*

PROOF (Sketch) The fact that $(\mathsf{Enc}, \mathsf{Dec})$ is EAV-secure means that, for any $S \subseteq \{0,1\}^\ell$, no PPT adversary can distinguish between $\mathsf{Enc}_k(m)$ (for uniform $m \in S$) and $\mathsf{Enc}_k(1^\ell)$. 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 $\mathsf{Enc}_k(1^\ell)$ with almost the same probability; otherwise, $\mathcal{A}$ could be used to distinguish between $\mathsf{Enc}_k(m)$ and $\mathsf{Enc}_k(1^\ell)$. The distinguisher is easily constructed: choose uniform $m \in S$, and output $m_0 = m$, $m_1 = 1^\ell$. When given a ciphertext c that is an encryption of either m_0 or m_1, invoke $\mathcal{A}(1^n, c)$ and output 0 if and only if $\mathcal{A}$ outputs $f(m)$. If $\mathcal{A}$ outputs $f(m)$ when given an encryption of m with probability that is significantly different from the probability that it outputs $f(m)$ when given an encryption of 1^ℓ, then the described distinguisher violates Definition 3.9.

The above suggests the following algorithm $\mathcal{A}'$ that does not receive $c = \mathsf{Enc}_k(m)$, yet computes $f(m)$ almost as well as $\mathcal{A}$ does: $\mathcal{A}'(1^n)$ chooses a uniform key $k \in \{0,1\}^n$, invokes $\mathcal{A}$ on $c \leftarrow \mathsf{Enc}_k(1^\ell)$, and outputs whatever $\mathcal{A}$ does. By the above, we have that $\mathcal{A}$ outputs $f(m)$ when run as a subroutine 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 guarantees considerably more than the property considered in Theorem 3.11. The definition allows the length of the plaintext to depend on the security parameter, and allows for essentially arbitrary distributions over plaintexts. (Actually, we allow only *efficiently sampleable* distributions. This means that there is some probabilistic polynomial-time algorithm Samp such that $\mathsf{Samp}(1^n)$ outputs messages according to the distribution.) The definition also takes into account arbitrary "external" information $h(m)$ about the plaintext that may be leaked to the adversary through other means (e.g., because the same message m is used for some other purpose as well).

DEFINITION 3.12 *A private-key encryption scheme* $(\mathsf{Enc}, \mathsf{Dec})$ *is* semantically secure in the presence of an eavesdropper *if for every* PPT *algorithm* $\mathcal{A}$ *there exists a* PPT *algorithm* $\mathcal{A}'$ *such that for any* PPT *algorithm* Samp *and polynomial-time computable functions* f *and* h*, the following is negligible:*

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

where the first probability is taken over uniform $k \in \{0,1\}^n$*,* m *output by* $\mathsf{Samp}(1^n)$*, the randomness of* $\mathcal{A}$*, and the randomness of* Enc*, and the second probability is taken over* m *output by* $\mathsf{Samp}(1^n)$ *and the randomness of* $\mathcal{A}'$*.*

The adversary $\mathcal{A}$ is given the ciphertext $\mathsf{Enc}_k(m)$ as well as the external information $h(m)$, and attempts to guess the value of $f(m)$. Algorithm $\mathcal{A}'$ also attempts to guess the value of $f(m)$, but is given *only* $h(m)$ and the

length of m. The security requirement states that $\mathcal{A}$'s probability of correctly guessing $f(m)$ is about the same as that of $\mathcal{A}'$. Intuitively, then, the ciphertext $\mathsf{Enc}_k(m)$ does not reveal any additional information about the value of $f(m)$.

Definition 3.12 constitutes a very strong and convincing formulation of the security guarantees that should be provided by an encryption scheme. However, it is easier to work with the definition of indistinguishability (Definition 3.8). Fortunately, the definitions are *equivalent*:

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

Looking ahead, a similar equivalence between semantic security and indistinguishability is known for all the definitions that we present in this chapter as well as those in Chapter 11. We can therefore use indistinguishability as our working definition, while being assured that the guarantees achieved are those of semantic security.

3.3 Constructing Secure Encryption Schemes

Having defined what it means for an encryption scheme to be secure, the reader may expect us to turn immediately to constructions of secure encryption schemes. Before doing so, however, we need to introduce the notions of *pseudorandom generators* (PRGs) and *stream ciphers*, important building blocks for private-key encryption. These, in turn, will lead to a discussion of *pseudorandomness*, which plays a fundamental role in cryptography in general and private-key encryption in particular.

3.3.1 Pseudorandom Generators and Stream Ciphers

A pseudorandom generator G is an efficient, deterministic algorithm for transforming a short, uniform string called the *seed* into a longer, "uniform-looking" (or "pseudorandom") output string. Stated differently, a pseudorandom generator uses a small amount of true randomness in order to generate a large amount of pseudorandomness. This is useful whenever a large number of random(-looking) bits are needed, since generating true random bits is difficult and slow. (See the discussion at the beginning of Chapter 2.) Indeed, pseudorandom generators have been studied since at least the 1940s when they were proposed for running statistical simulations. In that context, researchers proposed various statistical tests that a pseudorandom generator should pass in order to be considered "good." As a simple example, the first

bit of the output of a pseudorandom generator should be equal to 1 with probability very close to 1/2 (where the probability is taken over uniform choice of the seed), since the first bit of a uniform string is equal to 1 with probability exactly 1/2. In fact, the parity of any fixed subset of the output bits should also be 1 with probability very close to 1/2. More complex statistical tests can also be considered.

This historical approach to determining the quality of some candidate pseudorandom generator is ad hoc, and it is not clear when passing some set of statistical tests is sufficient to guarantee the soundness of using a candidate pseudorandom generator for some application. (In particular, there may be another statistical test that *does* successfully distinguish the output of the generator from true random bits.) The historical approach is even more problematic when using pseudorandom generators for cryptographic applications; in that setting, security may be compromised if an attacker is able to distinguish the output of a generator from uniform, and we do not know in advance what strategy an attacker might use.

The above considerations motivated a cryptographic approach to defining pseudorandom generators in the 1980s. The basic realization was that a good pseudorandom generator should pass *all* (efficient) statistical tests. That is, for *any* efficient statistical test (or *distinguisher*) D, the probability that D returns 1 when given the output of the pseudorandom generator should be close to the probability that D returns 1 when given a uniform string of the same length. Informally, then, the output of a pseudorandom generator should "look like" a uniform string to *any* efficient observer.

(We stress that, formally speaking, it does not make sense to say that any fixed string is "pseudorandom," in the same way that it is meaningless to refer to any fixed string as "random." Rather, pseudorandomness is a property of a *distribution* on strings. Nevertheless, we sometimes informally call a string sampled according to the uniform distribution a "uniform string," and a string output by a pseudorandom generator a "pseudorandom string.")

Another perspective is obtained by defining what it means for a distribution to be pseudorandom. Let Dist be a distribution on ℓ-bit strings. (This means that Dist assigns some probability to every string in $\{0,1\}^\ell$; sampling from Dist means that we choose an ℓ-bit string according to this probability distribution.) Informally, Dist is *pseudorandom* if the experiment in which a string is sampled from Dist is indistinguishable from the experiment in which a uniform string of length ℓ is sampled. (Strictly speaking, since we are in an asymptotic setting we need to speak of the pseudorandomness of a *sequence* of distributions $\mathsf{Dist} = \{\mathsf{Dist}_n\}$, where distribution Dist_n is used for security parameter n. We ignore this point in our current discussion.) More precisely, it should be infeasible for any polynomial-time algorithm to tell (better than guessing) whether it is given a string sampled according to Dist, or whether it is given a uniform ℓ-bit string. This means that *a pseudorandom string is just as good as a uniform string*, as long as we consider only polynomial-time observers. Just as indistinguishability is a computational relaxation of

perfect secrecy, pseudorandomness is a computational relaxation of true randomness. (We will generalize this perspective when we discuss the notion of indistinguishability in Chapter 7.)

Now let $G : \{0,1\}^n \to \{0,1\}^\ell$ be a function, and define Dist to be the distribution on ℓ-bit strings obtained by choosing a uniform $s \in \{0,1\}^n$ and outputting $G(s)$. Then G is a pseudorandom generator if and only if the distribution Dist is pseudorandom.

The formal definition. As discussed above, G is a pseudorandom generator if no efficient distinguisher can detect whether it is given a string output by G or a string chosen uniformly at random. As in Definition 3.9, this is formalized by requiring that every efficient algorithm outputs 1 with almost the same probability when given $G(s)$ (for uniform seed s) or a uniform string. (For an equivalent definition analogous to Definition 3.8, see Exercise 3.5.) We obtain a definition in the asymptotic setting by letting the security parameter n determine the length of the seed. We then insist that G be computable by an efficient algorithm. As a technicality, we also require that G's output be longer than its input; otherwise, G is not very useful or interesting.

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

1. **(Expansion:)** *For every n it holds that $\ell(n) > n$.*
2. **(Pseudorandomness:)** *For any* PPT *algorithm D, there is a negligible function* negl *such that*

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

where the first probability is taken over uniform choice of $s \in \{0,1\}^n$ and the randomness of D, and the second probability is taken over uniform choice of $r \in \{0,1\}^{\ell(n)}$ and the randomness of D.

We call ℓ the expansion factor *of G.*

We give an example of an insecure pseudorandom generator to gain familiarity with the definition.

Example 3.15
Define $G(s)$ to output s followed by $\oplus_{i=1}^{n} s_i$, so the expansion factor of G is $\ell(n) = n + 1$. The output of G can easily be distinguished from uniform. Consider the following efficient distinguisher D: on input a string w, output 1 if and only if the final bit of w is equal to the XOR of all the preceding bits of w. Since this property holds for all strings output by G, we have

$\Pr[D(G(s)) = 1] = 1$. On the other hand, if w is uniform, the final bit of w is uniform and so $\Pr[D(w) = 1] = \frac{1}{2}$. The quantity $|\frac{1}{2} - 1|$ is constant, not negligible, and so this G is not a pseudorandom generator. (Note that D is not always "correct," since it sometimes outputs 1 even when given a uniform string. This does not change the fact that D is a good distinguisher.) $\diamondsuit$

Discussion. The distribution on the output of a pseudorandom generator G is far from uniform. To see this, consider the case that $\ell(n) = 2n$ and so G doubles the length of its input. Under the uniform distribution on $\{0,1\}^{2n}$, each of the 2^{2n} possible strings is chosen with probability exactly 2^{-2n}. In contrast, consider the distribution of the output of G (when G is run on a uniform seed). When G receives an input of length n, the number of different strings in the range of G is at most 2^n. The fraction of strings of length $2n$ that are in the range of G is thus at most $2^n/2^{2n} = 2^{-n}$, and we see that the vast majority of 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*. Let G be as above and consider the exponential-time distinguisher D that works as follows: $D(w)$ outputs 1 if and only if there exists an $s \in \{0,1\}^n$ such that $G(s) = w$. (This computation is carried out in exponential time by exhaustively computing $G(s)$ for every $s \in \{0,1\}^n$. Recall that by Kerckhoffs' principle, the specification of G is known to D.) Now, if w were output by G, then 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}. So

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

which is large. This is just another example of a *brute-force attack*, and does not contradict the pseudorandomness of G since the attack is not efficient.

The seed and its length. The seed for a pseudorandom generator is analogous to the cryptographic key used by an encryption scheme, and the seed must be chosen uniformly and be kept secret from any adversary. Another important point, evident from the above discussion of brute-force attacks, is that s must be long enough so that it is not feasible to enumerate all possible seeds. In an asymptotic sense this is taken care of by setting the length of the seed equal to the security parameter, so that exhaustive search over all possible seeds requires exponential time. In practice, the seed must be long enough so that it is impossible to try all possible seeds within some specified time bound.

On the existence of pseudorandom generators. Do pseudorandom generators exist? They certainly seem difficult to construct, and one may rightly ask whether any algorithm satisfying Definition 3.14 exists. Although we do not know how to unconditionally prove the existence of pseudorandom generators, we have strong reasons to believe they exist. For one, they can be

constructed under the rather weak assumption that *one-way functions* exist (which is true if certain problems like factoring large numbers are hard); this will be discussed in detail in Chapter 7. We also have several practical constructions of candidate pseudorandom generators called *stream ciphers* for which no efficient distinguishers are known. (Later, we will introduce even stronger primitives called *block ciphers*.) We give a high-level overview of stream ciphers next, and discuss concrete stream ciphers in Chapter 6.

Stream Ciphers

Our definition of a pseudorandom generator is limited in two ways: the expansion factor is fixed, and the generator produces its entire output in "one shot." *Stream ciphers*, used in practice to instantiate pseudorandom generators, work somewhat differently. The pseudorandom output bits of a stream cipher are produced gradually and on demand, so that an application can request exactly as many pseudorandom bits as needed. This improves efficiency (since an application can request fewer bits, if sufficient) and flexibility (since there is no upper bound on the number of bits that can be requested).

Formally, we view a stream cipher[2] as a pair of deterministic algorithms $(\mathsf{Init}, \mathsf{GetBits})$ where:

- Init takes as input a seed s and an optional *initialization vector* IV, and outputs an initial state st_0.
- $\mathsf{GetBits}$ takes as input state information st_i, and outputs a bit y and updated state st_{i+1}. (In practice, y is a *block* of several bits; we treat y as a single bit here for generality and simplicity.)

Given a stream cipher and any desired expansion factor ℓ, we can define an algorithm G_ℓ mapping inputs of length n to outputs of length $\ell(n)$. The algorithm simply runs Init, and then repeatedly runs $\mathsf{GetBits}$ a total of ℓ times.

ALGORITHM 3.16
Constructing G_ℓ from $(\mathsf{Init}, \mathsf{GetBits})$

Input: Seed s and optional initialization vector IV
Output: $y_1, \ldots, y_\ell$

$\mathsf{st}_0 := \mathsf{Init}(s, IV)$
for $i = 1$ to ℓ:
  $(y_i, \mathsf{st}_i) := \mathsf{GetBits}(\mathsf{st}_{i-1})$
return $y_1, \ldots, y_\ell$

[2]The terminology here is not completely standard, and beware that "stream cipher" is used by different people in different (but related) ways. For example, some use it to refer to G_ℓ (see below), while some use it to refer to Construction 3.17 when instantiated with G_ℓ.

A stream cipher is *secure* in the basic sense if it takes no IV and for any polynomial ℓ with $\ell(n) > n$, the function G_ℓ constructed above is a pseudorandom generator with expansion factor ℓ. We briefly discuss one possible security notion for stream ciphers that use an IV in Section 3.6.1.

3.3.2 Proofs by Reduction

If we wish to prove that a given construction is computationally secure, then we must rely on unproven assumptions[3] (unless the scheme is information-theoretically secure). Our strategy will be to assume that some mathematical problem is hard, or that some *low-level* cryptographic primitive is secure, and then to *prove* that a given construction based on this problem/primitive is secure under this assumption. In Section 1.4.2 we have already explained in great detail why this approach is preferable so we do not repeat those arguments here.

The proof that a cryptographic construction is secure as long as some underlying problem is hard generally proceeds by presenting an explicit *reduction* showing how to transform any efficient adversary $\mathcal{A}$ that succeeds in "breaking" the construction into an efficient algorithm $\mathcal{A}'$ that solves the problem that was assumed to be hard. Since this is so important, we walk through a high-level outline of the steps of such a proof in detail. (We will see numerous concrete examples through the book, beginning with the proof of Theorem 3.18.) 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). A proof proceeds via the following steps (see also Figure 3.1):

1. Fix some efficient (i.e., probabilistic polynomial-time) adversary $\mathcal{A}$ attacking Π. Denote this adversary's success probability by $\varepsilon(n)$.
2. Construct an efficient algorithm $\mathcal{A}'$, called the "reduction," that attempts to solve problem X using adversary $\mathcal{A}$ as a subroutine. An important point here is that $\mathcal{A}'$ knows nothing about how $\mathcal{A}$ works; the only thing $\mathcal{A}'$ knows is that $\mathcal{A}$ is expecting to attack Π. So, given some input instance x of problem X, our algorithm $\mathcal{A}'$ will *simulate* for $\mathcal{A}$ an instance of Π such that:
   (a) As far as $\mathcal{A}$ can tell, it is interacting with Π. That is, the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ should be distributed identically to (or at least close to) the view of $\mathcal{A}$ when it interacts with Π itself.
   (b) If $\mathcal{A}$ succeeds in "breaking" the instance of Π that is being simulated by $\mathcal{A}'$, this should allow $\mathcal{A}'$ to solve the instance x it was given, at least with inverse polynomial probability $1/p(n)$.

[3] In particular, most of cryptography requires the unproven assumption that $\mathcal{P} \neq \mathcal{NP}$.

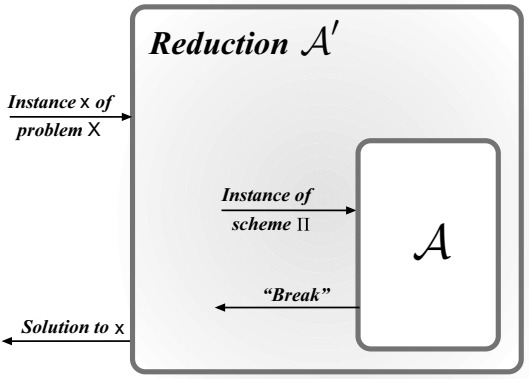


FIGURE 3.1: A high-level overview of a security proof by reduction.

3. Taken together, 2(a) and 2(b) imply that $\mathcal{A}'$ solves X with probability $\varepsilon(n)/p(n)$. If $\varepsilon(n)$ is not negligible, then neither is $\varepsilon(n)/p(n)$. Moreover, if $\mathcal{A}$ is efficient then we obtain an efficient algorithm $\mathcal{A}'$ solving X with non-negligible probability, contradicting the initial assumption.

4. Given our assumption regarding X, we conclude that *no* efficient adversary $\mathcal{A}$ can succeed in breaking Π with non-negligible probability. Stated differently, Π is computationally secure.

In the following section we will illustrate exactly the above idea: we will show how to use any pseudorandom generator G to construct an encryption scheme; we prove the encryption scheme secure by showing that any attacker who can "break" the encryption scheme can be used to distinguish the output of G from a uniform string. Under the assumption that G is a pseudorandom generator, then, the encryption scheme is secure.

3.3.3 A Secure Fixed-Length Encryption Scheme

A pseudorandom generator provides a natural way to construct a secure, fixed-length encryption scheme with a key shorter than the message. Recall that in the one-time pad (see Section 2.2), encryption is done by XORing a random pad with the message. The insight is that we can use a *pseudorandom* pad instead. Rather than sharing this long, pseudorandom pad, however, the sender and receiver can instead share a *seed* which is used to generate that pad when needed (see Figure 3.2); this seed will be shorter than the pad and hence shorter than the message. As for security, the intuition is that a pseudorandom string "looks random" to any polynomial-time adversary and so a computationally bounded eavesdropper cannot distinguish between a message encrypted using the one-time pad or a message encrypted using this "pseudo-"one-time pad encryption scheme.

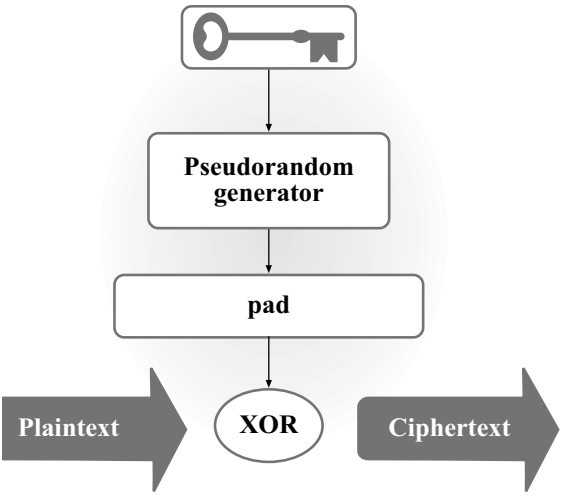


FIGURE 3.2: Encryption with a pseudorandom generator.

The encryption scheme. Fix some message length ℓ and 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 key-generation algorithm is the trivial one: $\mathsf{Gen}(1^n)$ simply outputs a uniform key k of length n. Encryption works by applying G to the key (which serves as a seed) in order to obtain a pad that is then XORed with the plaintext. Decryption applies G to the key and XORs the resulting pad with the ciphertext to recover the message. The scheme is described formally in Construction 3.17. In Section 3.6.1, we describe how stream ciphers are used to implement a variant of this scheme in practice.

CONSTRUCTION 3.17

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 uniform $k \in \{0,1\}^n$ 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 message
$$m := G(k) \oplus c.$$

A private-key encryption scheme based on any pseudorandom generator.

THEOREM 3.18 *If G is a pseudorandom generator, then Construction 3.17 is a fixed-length private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper.*

PROOF Let Π denote Construction 3.17. We show that Π satisfies Definition 3.8. Namely, we show that for any probabilistic polynomial-time adversary $\mathcal{A}$ there is a negligible function negl such that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n). \tag{3.2}$$

The intuition is that if Π used a uniform pad in place of the pseudorandom pad $G(k)$, 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 $1/2$. Thus, if Equation (3.2) does *not* hold then $\mathcal{A}$ must implicitly be distinguishing the output of G from a random string. We make this explicit by showing a *reduction*; namely, by showing how to use $\mathcal{A}$ to construct an efficient distinguisher D, with the property that D's ability to distinguish the output of G from a uniform string is directly related to $\mathcal{A}$'s ability to determine which message was encrypted by Π. Security of G then implies security of Π.

Let $\mathcal{A}$ be an arbitrary PPT adversary. We construct a distinguisher D that takes a string w as input, and whose goal is to determine whether w was chosen uniformly (i.e., w is a "random string") or whether w was generated by choosing a uniform k and computing $w := G(k)$ (i.e., w is a "pseudorandom string"). We construct D so that it emulates the eavesdropping experiment for $\mathcal{A}$, as described below, and observes whether $\mathcal{A}$ succeeds or not. If $\mathcal{A}$ succeeds then D guesses that w must be a pseudorandom string, while if $\mathcal{A}$ does not succeed then D guesses that w is a random string. In detail:

> **Distinguisher D:**
> D is given as input a string $w \in \{0,1\}^{\ell(n)}$. (We assume that n can be determined from $\ell(n)$.)
>
> 1. Run $\mathcal{A}(1^n)$ to obtain a pair of messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.
> 2. Choose a uniform bit $b \in \{0,1\}$. 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.

D clearly runs in polynomial time (assuming $\mathcal{A}$ does).

Before analyzing the behavior of D, we define a modified encryption scheme $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ that is exactly the one-time pad encryption scheme, except that we now incorporate a security parameter that determines the length of the message to be encrypted. That is, $\widetilde{\mathsf{Gen}}(1^n)$ outputs a uniform key k of length $\ell(n)$, and the encryption of message $m \in 2^{\ell(n)}$ using 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.) Perfect secrecy of the one-time pad implies

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

To analyze the behavior of D, the main observations are:

1. If w is chosen uniformly from $\{0,1\}^{\ell(n)}$, then the view of $\mathcal{A}$ when run as a subroutine 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 when $\mathcal{A}$ is run as a subroutine by $D(w)$ in this case, $\mathcal{A}$ is given a ciphertext $c = w \oplus m_b$ where $w \in \{0,1\}^{\ell(n)}$ is uniform. Since D outputs 1 exactly when $\mathcal{A}$ succeeds in its eavesdropping experiment, we therefore have (cf. Equation (3.3))

$$\Pr_{w \leftarrow \{0,1\}^{\ell(n)}}[D(w) = 1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] = \frac{1}{2}. \tag{3.4}$$

   (The subscript on the first probability just makes explicit that w is chosen uniformly from $\{0,1\}^{\ell(n)}$ there.)

2. If w is instead generated by choosing uniform $k \in \{0,1\}^n$ and then setting $w := G(k)$, the view of $\mathcal{A}$ when run as a subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. This is because $\mathcal{A}$, when run as a subroutine by D, is now given a ciphertext $c = w \oplus m_b$ where $w = G(k)$ for a uniform $k \in \{0,1\}^n$. Thus,

$$\Pr_{k \leftarrow \{0,1\}^n}[D(G(k)) = 1] = \Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right]. \tag{3.5}$$

Since G is a pseudorandom generator (and since D runs in polynomial time), we know there is a negligible function negl such that

$$\left|\Pr_{w \leftarrow \{0,1\}^{\ell(n)}}[D(w) = 1] - \Pr_{k \leftarrow \{0,1\}^n}[D(G(k)) = 1]\right| \leq \mathsf{negl}(n).$$

Using Equations (3.4) and (3.5), we thus see that

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

which implies $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)$. Since $\mathcal{A}$ was an arbitrary PPT adversary, this completes 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, the one-time pad also encrypts an ℓ-bit message by XORing it with an ℓ-bit string! The point of the construction, of course, is that the ℓ-bit string $G(k)$ can be *much*

longer than the shared key k. In particular, using the above scheme it is possible to securely encrypt a 1 Mb file using only a 128-bit key. By relying on computational secrecy we have thus circumvented the impossibility result of Theorem 2.10, which states that any *perfectly* secret encryption scheme must use a key at least as long as the message.

Reductions—a discussion. We do *not* prove unconditionally that Construction 3.17 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 higher-level construction to a lower-level primitive has a number of advantages (as discussed in Section 1.4.2). One of these advantages is that, in general, it is easier to design a lower-level primitive than a higher-level one; it is also easier, in general, to directly analyze an algorithm G with respect to a lower-level definition than to analyze a more complex scheme Π with respect to a higher-level definition. This does not mean that constructing a pseudorandom generator is "easy," only that it is easier than constructing an encryption scheme from scratch. (In the present case the encryption scheme does nothing except XOR the output of a pseudorandom generator with the message and so this isn't really true. However, we will see more complex constructions and in those cases the ability to reduce the task to a simpler one is of great importance.) Another advantage is that once an appropriate G has been constructed, it can be used as a component of various other schemes.

Concrete security. Although Theorem 3.18 and its proof are in an asymptotic setting, we can readily adapt the proof to bound the *concrete* security of the encryption scheme in terms of the concrete security of G. Fix some value of n for the remainder of this discussion, and let Π now denote Construction 3.17 using this value of n. Assume G is (t, ε)-pseudorandom (for the given value of n), in the sense that for all distinguishers D running in time at most t we have

$$\left| \Pr[D(r) = 1] - \Pr[D(G(s)) = 1] \right| \leq \varepsilon. \tag{3.6}$$

(Think of $t \approx 2^{80}$ and $\varepsilon \approx 2^{-60}$, though precise values are irrelevant for our discussion.) We claim that Π is $(t - c, \varepsilon)$-secure for some (small) constant c, in the sense that for all $\mathcal{A}$ running in time at most $t - c$ we have

$$\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A},\Pi} = 1\right] \leq \frac{1}{2} + \varepsilon. \tag{3.7}$$

(Note that the above are now fixed numbers, not functions of n, since we are not in an asymptotic setting here.) To see this, let $\mathcal{A}$ be an arbitrary adversary running in time at most $t - c$. Distinguisher D, as constructed in the proof of Theorem 3.18, has very little overhead besides running $\mathcal{A}$; setting c appropriately ensures that D runs in time at most t. Our assumption on the concrete security of G then implies Equation (3.6); proceeding exactly as in the proof of Theorem 3.18, we obtain Equation (3.7).

3.4 Stronger Security Notions

Until now we have considered a relatively weak definition of security in which the adversary only passively eavesdrops on a single ciphertext sent between the honest parties. In this section, we consider two stronger security notions. Recall that a security definition specifies a security goal and an attack model. In defining the first new security notion, we modify the security goal; for the second we strengthen the attack model.

3.4.1 Security for Multiple Encryptions

Definition 3.8 deals with the case where the communicating parties transmit a single ciphertext that is observed by an eavesdropper. It would be convenient, however, if the communicating parties could send multiple ciphertexts to each other—all generated using the same key—even if an eavesdropper might observe all of them. For such applications we need an encryption scheme secure for the encryption of multiple messages.

We begin with an appropriate definition of security for this setting. As in the case of Definition 3.8, we first introduce an appropriate experiment defined for any encryption scheme Π, adversary $\mathcal{A}$, and security parameter n:

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

1. *The adversary $\mathcal{A}$ is given input 1^n, and outputs a pair of equal-length* lists *of messages $\vec{M}_0 = (m_{0,1}, \ldots, m_{0,t})$ and $\vec{M}_1 = (m_{1,1}, \ldots, m_{1,t})$, with $|m_{0,i}| = |m_{1,i}|$ for all i.*
2. *A key k is generated by running $\mathsf{Gen}(1^n)$, and a uniform bit $b \in \{0,1\}$ is chosen. For all i, the ciphertext $c_i \leftarrow \mathsf{Enc}_k(m_{b,i})$ is computed and the list $\vec{C} = (c_1, \ldots, c_t)$ is given to $\mathcal{A}$.*
3. *$\mathcal{A}$ outputs a bit b'.*
4. *The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.*

The definition of security is the same as before, except that it now refers to the above experiment.

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 is 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 randomness used by $\mathcal{A}$ and the randomness used in the experiment.

Any scheme that has indistinguishable multiple encryptions in the presence of an eavesdropper clearly also satisfies Definition 3.8, since experiment $\mathsf{PrivK}^{\mathsf{eav}}$ corresponds to the special case of $\mathsf{PrivK}^{\mathsf{mult}}$ where the adversary outputs two lists containing only a single message each. In fact, our new definition is strictly stronger than Definition 3.8, as the following shows.

PROPOSITION 3.20 *There is a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper, but not indistinguishable* multiple *encryptions in the presence of an eavesdropper.*

PROOF We do not have to look far to find an example of an encryption scheme satisfying the proposition. The one-time pad is perfectly secret, and so also has indistinguishable encryptions in the presence of an eavesdropper. We show that it is not secure in the sense of Definition 3.19. (We have discussed this attack in Chapter 2 already; here, we merely analyze the attack with respect to Definition 3.19.)

Concretely, consider the following adversary $\mathcal{A}$ attacking the scheme (in the sense defined by experiment $\mathsf{PrivK}^{\mathsf{mult}}$): $\mathcal{A}$ outputs $\vec{M}_0 = (0^\ell, 0^\ell)$ and $\vec{M}_1 = (0^\ell, 1^\ell)$. (The first contains the same plaintext twice, while the second contains two different messages.) Let $\vec{C} = (c_1, c_2)$ be the list of ciphertexts that $\mathcal{A}$ receives. If $c_1 = c_2$, then $\mathcal{A}$ outputs $b' = 0$; otherwise, $\mathcal{A}$ outputs $b' = 1$.

We now analyze the probability that $b' = b$. The crucial point is that the one-time pad is *deterministic*, so encrypting the same message twice (using the same key) yields the same ciphertext. Thus, if $b = 0$ then we must have $c_1 = c_2$ and $\mathcal{A}$ outputs 0 in this case. On the other hand, if $b = 1$ then a different message is encrypted each time; hence $c_1 \neq c_2$ and $\mathcal{A}$ outputs 1. We conclude that $\mathcal{A}$ correctly outputs $b' = b$ with probability 1, and so the encryption scheme is not secure with respect to Definition 3.19. ■

Necessity of probabilistic encryption. The above might appear to show that Definition 3.19 is impossible to achieve using *any* encryption scheme. But in fact this is true only if the encryption scheme is *deterministic* and so encrypting the same message multiple times (using the same key) always yields the same result. This is important enough to state as a theorem.

THEOREM 3.21 *If* Π *is a (stateless*[4]*) encryption scheme in which* Enc *is a deterministic function of the key and the message, then* Π *cannot have indistinguishable multiple encryptions in the presence of an eavesdropper.*

This should not be taken to mean that Definition 3.19 is too strong. Indeed,

[4]We will see in Section 3.6.1 that if the encryption scheme is stateful, then it is possible to securely encrypt multiple messages even if encryption is deterministic.

leaking to an eavesdropper the fact that two encrypted messages are the same can be a significant security breach. (Consider, e.g., a scenario in which a student encrypts a series of true/false answers!)

To construct a scheme secure for encrypting multiple messages, we must design a scheme in which encryption is *randomized* so that when the same message is encrypted multiple times, different ciphertexts can be produced. This may seem impossible since decryption must always be able to recover the message. However, we will soon see how to achieve it.

3.4.2 Chosen-Plaintext Attacks and CPA-Security

Chosen-plaintext attacks capture the ability of an adversary to exercise (partial) control over what the honest parties encrypt. We imagine a scenario in which two honest parties share a key k, and the attacker can influence these parties to encrypt messages $m_1, m_2, \ldots$ (using k) and send the resulting ciphertexts over a channel that the attacker can observe. At some later point in time, the attacker observes a ciphertext corresponding to some *unknown* message m encrypted using the same key k; let us even assume that the attacker knows that m is one of two possibilities m_0, m_1. Security against chosen-plaintext attacks means that even in this case the attacker cannot tell which of these two messages was encrypted with probability significantly better than random guessing. (For now we revert back to the case where the eavesdropper is given only a single encryption of an unknown message. Shortly, we will return to consideration of the multiple-message case.)

Chosen-plaintext attacks in the real world. Are chosen-plaintext attacks a realistic concern? For starters, note that chosen-plaintext attacks also encompass *known-plaintext attacks*—in which the attacker knows what messages are being encrypted, even if it does not get to choose them—as a special case. Moreover, there are several real-world scenarios in which an adversary might have significant influence over what messages get encrypted. A simple example is given by an attacker typing on a terminal, which in turn encrypts and sends everything the adversary types using a key shared with a remote server (and unknown to the attacker). Here the attacker exactly controls what gets encrypted, but the encryption scheme should remain secure when it is used—with the same key— to encrypt data for another user.

Interestingly, chosen-plaintext attacks have also been used successfully as part of historical efforts to break military encryption schemes. For example, during World War II the British placed mines at certain locations, knowing that the Germans—when finding those mines—would encrypt the locations and send them back to headquarters. These encrypted messages were used by cryptanalysts at Bletchley Park to break the German encryption scheme.

Another example is given by the famous story involving the Battle of Midway. In May 1942, US Navy cryptanalysts intercepted an encrypted message from the Japanese which they were able to partially decode. The result in-

dicated that the Japanese were planning an attack on `AF`, where `AF` was a ciphertext fragment that the US was unable to decode. For other reasons, the US believed that Midway Island was the target. Unfortunately, their attempts to convince Washington planners that this was the case were futile; the general belief was that Midway could not possibly be the target. The Navy cryptanalysts devised the following plan: They instructed US forces at Midway to send a fake message that their freshwater supplies were low. The Japanese intercepted this message and immediately reported to their superiors that "`AF` is low on water." The Navy cryptanalysts now had their proof that `AF` corresponded to Midway, and the US dispatched three aircraft carriers to that location. The result was that Midway was saved, and the Japanese incurred significant losses. This battle was a turning point in the war between the US and Japan in the Pacific.

The Navy cryptanalysts here carried out a chosen-plaintext attack, as they were able to influence the Japanese (albeit in a roundabout way) to encrypt the word "Midway." 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)!

CPA-security. In the formal definition we model chosen-plaintext attacks by giving the adversary $\mathcal{A}$ access to an *encryption oracle*, viewed as a "black box" that encrypts messages of $\mathcal{A}$'s choice using a key k that is unknown to $\mathcal{A}$. That is, we imagine $\mathcal{A}$ has access to an "oracle" $\mathsf{Enc}_k(\cdot)$; when $\mathcal{A}$ *queries* this oracle by providing it with a 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 randomness each time it answers a query.) The adversary is allowed to interact with the encryption oracle adaptively, as many times as it likes.

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

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

1. *A key k is generated by running $\mathsf{Gen}(1^n)$.*
2. *The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\mathsf{Enc}_k(\cdot)$, and outputs a pair of messages m_0, m_1 of the same length.*
3. *A uniform bit $b \in \{0,1\}$ is chosen, and then a ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ is computed and given to $\mathcal{A}$.*
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 the former case, we say that $\mathcal{A}$ succeeds.*

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 is 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 randomness used by $\mathcal{A}$*, as well as the randomness used in the experiment.*

CPA-Security for Multiple Encryptions

Definition 3.22 can be extended to the case of multiple encryptions in the same way that Definition 3.8 is extended to give Definition 3.19, i.e., by using lists of plaintexts. Here, we take a different approach that is somewhat simpler and has the advantage of modeling attackers that can *adaptively* choose plaintexts to be encrypted, even after observing previous ciphertexts. In the present definition, we give the attacker access to a "left-or-right" oracle $\mathsf{LR}_{k,b}$ that, on input a pair of equal-length messages m_0, m_1, computes the ciphertext $c \leftarrow \mathsf{Enc}_k(m_b)$ and returns c. That is, if $b = 0$ then the adversary receives an encryption of the "left" plaintext, and if $b = 1$ then it receives an encryption of the "right" plaintext. Here, b is a random bit chosen at the beginning of the experiment, and as in previous definitions the goal of the attacker is to guess b. This generalizes the previous definition of multiple-message security (Definition 3.19) because instead of outputting the lists $(m_{0,1}, \ldots, m_{0,t})$ and $(m_{1,1}, \ldots, m_{1,t})$, one of whose messages will be encrypted, the attacker can now sequentially query $\mathsf{LR}_{k,b}(m_{0,1}, m_{1,1})$, $\ldots$, $\mathsf{LR}_{k,b}(m_{0,t}, m_{1,t})$. This also encompasses the attacker's access to an encryption oracle, since the attacker can simply query $\mathsf{LR}_{k,b}(m, m)$ to obtain $\mathsf{Enc}_k(m)$.

We now formally define this experiment, called the LR-oracle experiment. Let Π be an encryption scheme, $\mathcal{A}$ an adversary, and n the security parameter:

The LR-oracle experiment $\mathsf{PrivK}^{\mathsf{LR\text{-}cpa}}_{\mathcal{A},\Pi}(n)$**:**

1. *A key* k *is generated by running* $\mathsf{Gen}(1^n)$*.*
2. *A uniform bit* $b \in \{0,1\}$ *is chosen.*
3. *The adversary* $\mathcal{A}$ *is given input* 1^n *and oracle access to* $\mathsf{LR}_{k,b}(\cdot,\cdot)$*, as defined above.*
4. *The adversary* $\mathcal{A}$ *outputs a bit* b'*.*
5. *The output of the experiment is defined to be* 1 *if* $b' = b$*, and* 0 *otherwise. In the former case, we say that* $\mathcal{A}$ succeeds*.*

DEFINITION 3.23 *Private-key encryption scheme* Π *has* indistinguishable multiple encryptions under a chosen-plaintext attack*, or is* CPA-secure for multiple encryptions*, if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there is a negligible function* negl *such that*

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

where the probability is taken over the randomness used by $\mathcal{A}$ *and the randomness used in the experiment.*

Our earlier discussion shows that CPA-security for multiple encryptions is at least as strong as all our previous definitions. In particular, if a private-key encryption scheme is CPA-secure for multiple encryptions then it is clearly CPA-secure as well. Importantly, the converse also holds; that is, CPA-security implies CPA-security for multiple encryptions. (This stands in contrast to the case of eavesdropping adversaries; see Proposition 3.20.) We state the following theorem here without proof; a similar result in the public-key setting is proved in Section 11.2.2.

THEOREM 3.24 *Any private-key encryption scheme that is CPA-secure is also CPA-secure for* multiple *encryptions.*

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

Security against chosen-plaintext attacks is nowadays the minimal notion of security an encryption scheme should satisfy, though it is becoming more common to require even the stronger security properties discussed in Section 4.5.

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 loss of generality. In particular, given any CPA-secure *fixed-length* encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, it is possible to construct a CPA-secure encryption scheme $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ for *arbitrary-length* messages quite easily. For simplicity, say Π encrypts messages that are 1-bit long (though everything we say extends in the natural way regardless of the message length supported by Π). Leave Gen' the same as Gen. Define Enc'_k for any message m (having some arbitrary length ℓ) as $\mathsf{Enc}'_k(m) = \mathsf{Enc}_k(m_1), \ldots, \mathsf{Enc}_k(m_\ell)$, where m_i denotes the ith bit of m. Decryption is done in the natural way. Π' is CPA-secure if Π is; a proof follows from Theorem 3.24.

There are more efficient ways to encrypt messages of arbitrary length than by adapting a fixed-length encryption scheme in the above manner. We explore this further in Section 3.6.

3.5 Constructing CPA-Secure Encryption Schemes

Before constructing encryption schemes secure against chosen-plaintext attacks, we first introduce the important notion of *pseudorandom functions*.

3.5.1 Pseudorandom Functions and Block Ciphers

Pseudorandom functions (PRFs) generalize the notion of pseudorandom generators. Now, instead of considering "random-looking" *strings* we consider "random-looking" *functions*. As in our earlier discussion of pseudorandomness, it does not make much sense to say that any *fixed* function $f : \{0,1\}^* \to \{0,1\}^*$ is pseudorandom (in the same way that it makes little sense to say that any fixed function is random). Thus, we must instead refer to the pseudorandomness of a *distribution* on functions. Such a distribution is induced naturally by considering *keyed functions*, defined next.

A keyed function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ is a two-input function, where the first input is called the *key* and denoted k. We say F is *efficient* if there is a polynomial-time algorithm that computes $F(k,x)$ given k and x. (We will only be interested in efficient keyed functions.) In typical usage a key k is chosen and fixed, and we are then interested in the single-input function $F_k : \{0,1\}^* \to \{0,1\}^*$ defined by $F_k(x) = F(k,x)$. The security parameter n dictates the key length, input length, and output length. That is, we associate with F three functions ℓ_{key}, ℓ_{in}, and ℓ_{out}; for any key $k \in \{0,1\}^{\ell_{key}(n)}$, the function F_k is only defined for inputs $x \in \{0,1\}^{\ell_{in}(n)}$, in which case $F_k(x) \in \{0,1\}^{\ell_{out}(n)}$. Unless stated otherwise, we assume for simplicity that F is *length-preserving*, meaning $\ell_{key}(n) = \ell_{in}(n) = \ell_{out}(n) = n$. That is, by fixing a key $k \in \{0,1\}^n$ we obtain a function F_k mapping n-bit input strings to n-bit output strings.

A keyed function F induces a natural distribution on functions given by choosing a uniform key $k \in \{0,1\}^n$ and then considering the resulting single-input function F_k. We call F *pseudorandom* if the function F_k (for a uniform 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 efficient adversary can distinguish—in a sense we more carefully define below—whether it is interacting with F_k (for uniform k) or f (where f is chosen uniformly from the set of all functions mapping n-bit inputs to n-bit outputs).

Since choosing a function at random is less intuitive than choosing a string at random, it is worth spending a bit more time on this idea. Consider the set Func_n of all functions mapping n-bit strings to n-bit strings. This set is finite, and selecting a uniform function mapping n-bit strings to n-bit strings means choosing an element uniformly from this set. How large is Func_n? A function f is specified by giving its value on each point in its domain. We can view any function (over a finite domain) as a large look-up table that stores

$f(x)$ in the row of the table labeled by x. For $f \in \mathsf{Func}_n$, the look-up table for f has 2^n rows (one for each point of the domain $\{0,1\}^n$), with each row containing an n-bit string (since the range of f is $\{0,1\}^n$). Concatenating all the entries of the table, we see that any function in Func_n can be represented by a string of length $2^n \cdot n$. Moreover, this correspondence is one-to-one, as each string of length $2^n \cdot n$ (i.e., each table containing 2^n entries of length n) defines a unique function in Func_n. Thus, the size of Func_n is exactly the number of strings of length $n \cdot 2^n$, or $|\mathsf{Func}_n| = 2^{n \cdot 2^n}$.

Viewing a function as a look-up table provides another useful way to think about selecting a uniform function $f \in \mathsf{Func}_n$: It is exactly equivalent to choosing each row in the look-up table of f uniformly. This means, in particular, that the values $f(x)$ and $f(y)$ (for any two inputs $x \neq y$) are uniform and independent. We can view this look-up table being populated by random entries in advance, before f is evaluated on any input, or we can view entries of the table being chosen uniformly "on-the-fly," as needed, whenever f is evaluated on a new input on which f has not been evaluated before.

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

A first attempt at formalizing the notion of a pseudorandom function would be to proceed in the same way as in Definition 3.14. That is, we could require that every polynomial-time distinguisher D that receives a description of the pseudorandom function F_k outputs 1 with "almost" the same probability as when it receives a description of a random function f. However, this definition is inappropriate since the description of a random function has *exponential length* (given by its look-up table of 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 definition therefore gives D access to an *oracle* $\mathcal{O}$ which is either equal to F_k (for uniform k) or f (for a uniform function f). The distinguisher D may query its oracle at any point x, in response to which the oracle returns $\mathcal{O}(x)$. We treat the oracle as a black box in the same way as when we provided the adversary with oracle access to the encryption algorithm in the definition of a chosen-plaintext attack. Here, however, the oracle computes a deterministic function and so returns the same result if queried twice on the same input. (For this reason, we may assume without loss of generality that D never queries the oracle twice on the same input.) D may interact freely with its oracle, choosing its queries adaptively based on all previous outputs. Since D runs in polynomial time, however, it can ask only polynomially many queries.

We now present the formal definition. (The definition assumes F is length-preserving for simplicity only.)

DEFINITION 3.25 *Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be an efficient, length-preserving, keyed function. F is a* pseudorandom function *if for all probabilistic polynomial-time distinguishers D, there is a negligible function* negl *such that:*

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

where the first probability is taken over uniform choice of $k \in \{0,1\}^n$ and the randomness of D, and the second probability is taken over uniform choice of $f \in \mathsf{Func}_n$ and the randomness of D.

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

Example 3.26
As usual, we can gain familiarity with the definition by looking at an insecure example. Define the keyed, length-preserving function F by $F(k,x) = k \oplus x$. For any input x, the value of $F_k(x)$ is uniformly distributed (when k is uniform). Nevertheless, F is not pseudorandom since its values on any *two* points are correlated. Concretely, consider the distinguisher D that queries its oracle $\mathcal{O}$ on arbitrary, distinct points x_1, x_2 to obtain values $y_1 = \mathcal{O}(x_1)$ and $y_2 = \mathcal{O}(x_2)$, and outputs 1 if and only if $y_1 \oplus y_2 = x_1 \oplus x_2$. If $\mathcal{O} = F_k$, for any k, then D outputs 1. On the other hand, if $\mathcal{O} = f$ for f chosen uniformly from Func_n, then the probability that $f(x_1) \oplus f(x_2) = x_1 \oplus x_2$ is exactly the probability that $f(x_2) = x_1 \oplus x_2 \oplus f(x_1)$, or 2^{-n}, and D outputs 1 with this probability. The difference is $|1 - 2^{-n}|$, which is not negligible. ♢

Pseudorandom Permutations/Block Ciphers

Let Perm_n be the set of all permutations (i.e., bijections) on $\{0,1\}^n$. Viewing any $f \in \mathsf{Perm}_n$ as a look-up table as before, we now have the added constraint that the entries in any two distinct rows must be different. We have 2^n different choices for the entry in the first row of the table; once we fix this entry, we are left with only $2^n - 1$ choices for the second row, and so on. We thus see that the size of Perm_n is $(2^n)!$.

Let F be a keyed function. We call F a *keyed permutation* if $\ell_{in} = \ell_{out}$, and furthermore for all $k \in \ell_{key}(n)$ the function $F_k : \{0,1\}^{\ell_{in}(n)} \to \{0,1\}^{\ell_{in}(n)}$ is one-to-one (i.e., F_k is a permutation). We call ℓ_{in} the *block length* of F. As before, unless stated otherwise we assume F is *length-preserving* and so $\ell_{key}(n) = \ell_{in}(n) = n$. A keyed permutation is *efficient* if there is a polynomial-time algorithm for computing $F_k(x)$ given k and x, as well as a polynomial-time algorithm for computing $F_k^{-1}(y)$ given k and y. That is, F_k should be both efficiently computable and *efficiently invertible* given k.

The definition of what it means for an efficient, keyed permutation F to be a *pseudorandom permutation* is exactly analogous to Definition 3.25, with the only difference being that now we require F_k to be indistinguishable from a uniform *permutation* rather than a uniform function. That is, we require that no efficient algorithm can distinguish between access to F_k (for uniform key k) and access to f (for uniform $f \in \mathsf{Perm}_n$). It turns out that this is merely an aesthetic choice since, whenever the block length is sufficiently long, a random permutation is itself indistinguishable from a random function. Intuitively this is due to the fact that a uniform function f looks identical to a uniform permutation unless distinct values x and y are found for which $f(x) = f(y)$, since in such a case the function cannot be a permutation. However, the probability of finding such values x, y using a polynomial number of queries is negligible. (This follows from the results of Appendix A.4.)

PROPOSITION 3.27 *If F is a pseudorandom permutation and additionally $\ell_{in}(n) \geq n$, then F is also a pseudorandom function.*

If F is a keyed permutation then cryptographic schemes based on F might require the honest parties to compute the inverse F_k^{-1} in addition to computing F_k itself. This potentially introduces new security concerns. In particular, it may now be necessary to impose the stronger requirement that F_k be indistinguishable from a uniform permutation *even if the distinguisher is additionally given oracle access to the inverse of the permutation.* If F has this property, we call it a *strong* pseudorandom permutation.

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

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

where the first probability is taken over uniform choice of $k \in \{0,1\}^n$ and the randomness of D, and the second probability is taken over uniform choice of $f \in \mathsf{Perm}_n$ and the randomness of D.

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

Block ciphers. In practice, *block ciphers* are designed to be secure instantiations of (strong) pseudorandom permutations with some *fixed* key length and block length. We discuss approaches for building block ciphers, and some popular candidate block ciphers, in Chapter 6. For the purposes of the present chapter the details of these constructions are unimportant, and for now we simply assume that (strong) pseudorandom permutations exist.

Pseudorandom functions and pseudorandom generators. As one might expect, there is a close relationship between pseudorandom functions and pseudorandom generators. It is fairly easy to construct a pseudorandom generator G from a pseudorandom function F by simply evaluating F on a series of different inputs; e.g., we can define $G(s) \stackrel{\text{def}}{=} F_s(1)\|F_s(2)\|\cdots\|F_s(\ell)$ for any desired ℓ. If F_s were replaced by a uniform function f, the output of G would be uniform; thus, when using F instead, the output is pseudorandom. You are asked to prove this formally in Exercise 3.14.

More generally, we can use the above idea to construct a stream cipher $(\mathsf{Init}, \mathsf{GetBits})$ that accepts an initialization vector IV. (See Section 3.3.1.) The only difference is that instead of evaluating F_s on the fixed input sequence $1, 2, 3, \ldots$, we evaluate F on the inputs $IV + 1$, $IV + 2$, $\ldots$.

CONSTRUCTION 3.29

Let F be a pseudorandom function. Define a stream cipher $(\mathsf{Init}, \mathsf{GetBits})$, where each call to $\mathsf{GetBits}$ outputs n bits, as follows:

- Init: on input $s \in \{0,1\}^n$ and $IV \in \{0,1\}^n$, set $\mathsf{st}_0 := (s, IV)$.
- $\mathsf{GetBits}$: on input $\mathsf{st}_i = (s, IV)$, compute $IV' := IV + 1$ and set $y := F_s(IV')$ and $\mathsf{st}_{i+1} := (s, IV')$. Output (y, st_{i+1}).

A stream cipher from any pseudorandom function/block cipher.

Although stream ciphers can be constructed from block ciphers, dedicated stream ciphers used in practice typically have better performance, especially in resource-constrained environments. On the other hand, stream ciphers appear to be less well understood (in practice) than block ciphers, and confidence in their security is lower. It is therefore recommended to use block ciphers (possibly by converting them to stream ciphers first) whenever possible.

Considering the other direction, a pseudorandom generator G immediately gives a pseudorandom function F *with small block length*. Specifically, say G has expansion factor $n \cdot 2^{t(n)}$. We can define the keyed function $F : \{0,1\}^n \times \{0,1\}^{t(n)} \to \{0,1\}^n$ as follows: to compute $F_k(i)$, first compute $G(k)$ and interpret the result as a look-up table with $2^{t(n)}$ rows each containing n bits; output the ith row. This runs in polynomial time only if $t(n) = \mathcal{O}(\log n)$. It is possible, though more difficult, to construct pseudorandom functions *with large block length* from pseudorandom generators; this is

shown in Section 7.5. Pseudorandom generators, in turn, can be constructed based on certain mathematical problems conjectured to be hard. The existence of pseudorandom functions based on these hard mathematical problems represents one of the amazing contributions of modern cryptography.

3.5.2 CPA-Secure Encryption from Pseudorandom Functions

We focus here on constructing a CPA-secure, fixed-length encryption scheme. By what we have said at the end of Section 3.4.2, this implies the existence of a CPA-secure encryption scheme for arbitrary-length messages. In Section 3.6 we will discuss more efficient ways of encrypting messages of arbitrary length.

A naive attempt at constructing a secure encryption scheme from a pseudorandom permutation is to define $\mathsf{Enc}_k(m) = F_k(m)$. Although we expect that this "reveals no information about m" (since, if f is a uniform function, then $f(m)$ is simply a uniform n-bit string), this method of encryption is *deterministic* and so cannot possibly be CPA-secure. In particular, encrypting the same plaintext twice will yield the same ciphertext.

Our secure construction is *randomized*. Specifically, we encrypt by applying the pseudorandom function to a *random value* r (rather than the message) and XORing the result with the plaintext. (See Figure 3.3 and Construction 3.30.) This can again be viewed as an instance of XORing a pseudorandom pad with the plaintext, with the major difference being the fact that a *fresh* pseudorandom pad is used each time. (In fact, the pseudorandom pad is only "fresh" if the pseudorandom function is applied to a "fresh" value on which it has never been applied before. While it is possible that a random r will be equal to some r-value chosen previously, this happens with only negligible probability.)

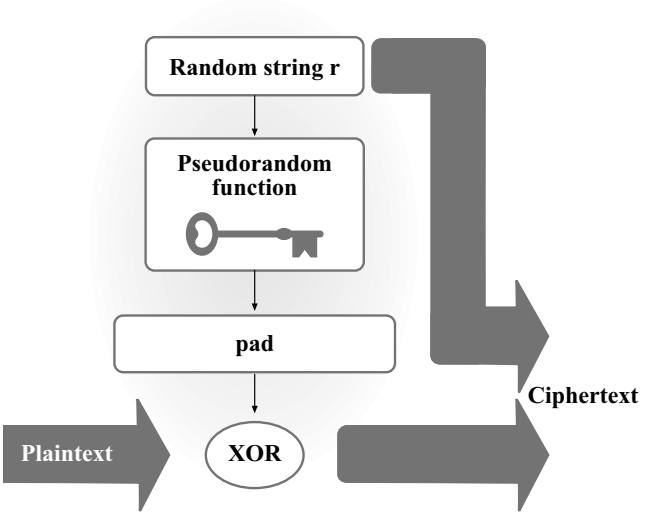


FIGURE 3.3: Encryption with a pseudorandom function.

Proofs of security based on pseudorandom functions. Before turning to the proof that the above construction is CPA-secure, we highlight a common template that is used by most proofs of security (even outside the context of encryption) for constructions based on pseudorandom functions. The first step of such proofs is to consider a hypothetical version of the construction in which the pseudorandom function is replaced with a random function. It is then argued—using a proof by reduction—that this modification does not significantly affect the attacker's success probability. We are then left with analyzing a scheme that uses a completely random function. At this point the rest of the proof typically relies on probabilistic analysis and does not rely on any computational assumptions. We will utilize this proof template several times in this and the next chapter.

CONSTRUCTION 3.30

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 uniform $k \in \{0,1\}^n$ and output it.
- Enc: on input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^n$, choose uniform $r \in \{0,1\}^n$ 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.

THEOREM 3.31 *If F is a pseudorandom function, then Construction 3.30 is a CPA-secure private-key encryption scheme for messages of length n.*

PROOF Let $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ be an encryption scheme that is exactly the same as $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ from Construction 3.30, except that a truly random function f is used in place of F_k. That is, $\widetilde{\mathsf{Gen}}(1^n)$ chooses a uniform function $f \in \mathsf{Func}_n$, and $\widetilde{\mathsf{Enc}}$ encrypts just like Enc except that f is used instead of F_k. (This modified encryption scheme is not efficient. But we can still define it as a hypothetical encryption scheme for the sake of the proof.)

Fix an arbitrary PPT adversary $\mathcal{A}$, and let $q(n)$ be an upper bound on the number of queries that $\mathcal{A}(1^n)$ makes to its encryption oracle. (Note that q must be upper-bounded by some polynomial.) As the first step of the proof, we show that there is a negligible function negl such that

$$\left| \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right] - \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] \right| \leq \mathsf{negl}(n). \tag{3.8}$$

We prove this by 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 $\mathcal{O}$, and its goal is to determine whether this function is "pseudorandom" (i.e., equal to F_k for uniform $k \in \{0,1\}^n$) or "random" (i.e., equal to f for uniform $f \in \mathsf{Func}_n$). To do this, D emulates experiment $\mathsf{PrivK}^{\mathsf{cpa}}$ for $\mathcal{A}$ in the manner described below, and observes whether $\mathcal{A}$ succeeds or not. If $\mathcal{A}$ succeeds then D guesses that its oracle must be a pseudorandom function, whereas if $\mathcal{A}$ does not succeed then D guesses that its oracle must be a random function. In detail:

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

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

D runs in polynomial time since $\mathcal{A}$ does. 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 subroutine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n)$. This is because, in this case, a key k is chosen uniformly at random and then every encryption is carried out by choosing a uniform r, computing $y := F_k(r)$, and setting the ciphertext equal to $\langle r, y \oplus m \rangle$, exactly as in Construction 3.30. Thus,

$$\Pr_{k \leftarrow \{0,1\}^n}\left[D^{F_k(\cdot)}(1^n) = 1\right] = \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right], \tag{3.9}$$

   where we emphasize that k is chosen uniformly on the left-hand side.

2. If D's oracle is a random function, then the view of $\mathcal{A}$ when run as a subroutine 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 uniform function $f \in \mathsf{Func}_n$ is used instead of F_k. Thus,

$$\Pr_{f \leftarrow \mathsf{Func}_n}\left[D^{f(\cdot)}(1^n) = 1\right] = \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right], \tag{3.10}$$

   where f is chosen uniformly from Func_n on the left-hand side.

By the assumption that F is a pseudorandom function (and since D is efficient), there exists a negligible function negl for which

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

Combining the above with Equations (3.9) and (3.10) gives Equation (3.8).

For the second part of the proof, we show that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)=1\right]\leq\frac{1}{2}+\frac{q(n)}{2^n}. \tag{3.11}$$

(Recall that $q(n)$ is a bound on the number of encryption queries made by $\mathcal{A}$. The above holds even if we place no computational restrictions on $\mathcal{A}$.) To see that Equation (3.11) holds, observe that every time a message m is encrypted in $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)$ (either by the encryption oracle or when the challenge ciphertext is computed), a uniform $r\in\{0,1\}^n$ is chosen and the ciphertext is set equal to $\langle r, f(r)\oplus m\rangle$. Let r^* denote the random string used when generating the challenge ciphertext $\langle r^*, f(r^*)\oplus m_b\rangle$. There are two possibilities:

1. *The value r^* is never used when answering any of $\mathcal{A}$'s encryption-oracle queries:* In this case, $\mathcal{A}$ learns nothing about $f(r^*)$ from its interaction with the encryption oracle (since f is a truly random function). This means that, as far as $\mathcal{A}$ is concerned, the value $f(r^*)$ that is XORed with m_b is uniformly distributed and independent of the rest of the experiment, 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).

2. *The value r^* is used when answering at least one of $\mathcal{A}$'s encryption-oracle queries:* In this case, $\mathcal{A}$ may easily determine whether m_0 or m_1 was encrypted. This is so because if the encryption oracle ever returns a ciphertext $\langle r^*, s\rangle$ in response to a request to encrypt the message m, the adversary learns that $f(r^*)=s\oplus m$.

   However, since $\mathcal{A}$ makes at most $q(n)$ queries to its encryption oracle (and thus at most $q(n)$ values of r are used when answering $\mathcal{A}$'s encryption-oracle queries), and since r^* is chosen uniformly from $\{0,1\}^n$, the probability of this event is at most $q(n)/2^n$.

Let Repeat denote the event that r^* is used by the encryption oracle when answering at least one of $\mathcal{A}$'s queries. As just discussed, the probability of Repeat is at most $q(n)/2^n$, and the probability that $\mathcal{A}$ succeeds in $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}$ if Repeat does not occur is exactly $1/2$. Therefore:

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

Combining the above with Equation (3.8), we see that there is a negligible function negl such that $\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \frac{q(n)}{2^n} + \mathsf{negl}(n)$. Since q is polynomial, $\frac{q(n)}{2^n}$ is negligible. In addition, the sum of two negligible functions is negligible, and thus there exists a negligible function negl' such that $\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}'(n)$, completing the proof. ■

Concrete security. The above proof shows that

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

for some negligible function negl. The final term depends on the security of F as a pseudorandom function; it is a bound on the distinguishing probability of algorithm D (which has roughly the same running time as the adversary $\mathcal{A}$). The term $\frac{q(n)}{2^n}$ represents a bound on the probability that the value r^* used to encrypt the challenge ciphertext was used to encrypt some other message.

3.6 Modes of Operation

Modes of operation provide a way to securely (and efficiently) encrypt long messages using stream or block ciphers.

3.6.1 Stream-Cipher Modes of Operation

Construction 3.17 provides a way to construct an encryption scheme using a pseudorandom generator. That scheme has two main drawbacks. First, as presented, the length of the message to be encrypted must be fixed and known in advance. Second, the scheme is only EAV-secure, not CPA-secure.

Stream ciphers, which can be viewed as flexible pseudorandom generators, can be used to address these drawbacks. In practice, stream ciphers are used for encryption in two ways: *synchronized mode* and *unsynchronized mode*.

Synchronized mode. In this stateful encryption scheme, the sender and receiver must be *synchronized* in the sense that they know how much plaintext has been encrypted (resp., decrypted) so far. Synchronized mode is typically used in a single communication session between parties (see Section 4.5.3), where statefulness is acceptable and messages are received in order without being lost. The intuition here is that a long pseudorandom stream is generated, and a different part of it is used to encrypt each message. Synchronization is needed to ensure correct decryption (i.e., so the receiver knows which part of the stream was used to encrypt the next message), and to ensure that no portion of the stream is re-used. We describe this mode in detail next.

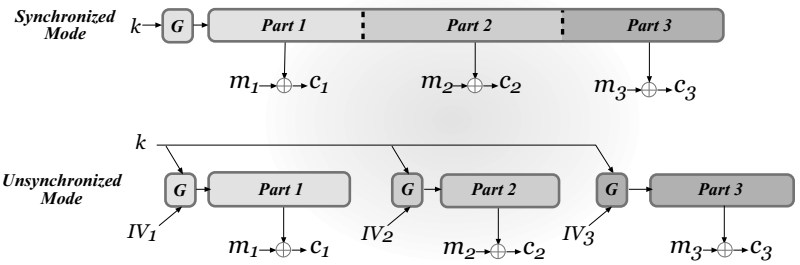


FIGURE 3.4: Synchronized mode and unsynchronized mode.

We have seen in Algorithm 3.16 that a stream cipher can be used to construct a pseudorandom generator G_ℓ with any desired expansion factor ℓ. We can easily modify that algorithm to obtain a pseudorandom generator G_∞ with *variable* output length. G_∞ takes two inputs: a seed s and a desired output length 1^ℓ (we specify this in unary since G_∞ will run in time polynomial in ℓ). As in Algorithm 3.16, $G_\infty(s, 1^\ell)$ runs $\mathsf{Init}(s)$ and then repeatedly runs $\mathsf{GetBits}$ a total of ℓ times.

We can use G_∞ in Construction 3.17 to handle encryption of arbitrary-length messages: encryption of a message m using the key k is done by computing the ciphertext $c := G_\infty(k, 1^{|m|}) \oplus m$; decryption of a ciphertext c using the key k is carried out by computing the message $m := G_\infty(k, 1^{|c|}) \oplus c$. A minor modification of the proof of Theorem 3.18 shows that if the stream cipher is secure then this encryption scheme has indistinguishable encryptions in the presence of an eavesdropper.

A little thought shows that if the communicating parties are willing to maintain state, then they can use the same key to encrypt *multiple* messages. (See Figure 3.4.) The conceptual insight is that the parties can treat multiple messages $m_1, m_2, \ldots$ as a single, long message; furthermore, Construction 3.17 (as well as the modified version in the previous paragraph) has the property that initial portions of a message can be encrypted and transmitted even if the rest of the message is not yet known. Concretely, the parties share a key k and both begin by computing $\mathsf{st}_0 := \mathsf{Init}(k)$. To encrypt the first message m_1 of length ℓ_1, the sender repeatedly runs $\mathsf{GetBits}$ a total of ℓ_1 times, beginning at st_0, to obtain a stream of bits $\mathsf{pad}_1 \stackrel{\text{def}}{=} y_1, \ldots, y_{\ell_1}$ along with updated state st_{ℓ_1}; it then sends $c_1 := \mathsf{pad}_1 \oplus m_1$. Upon receiving c_1, the other party repeatedly runs $\mathsf{GetBits}$ a total of ℓ_1 times to obtain the same values pad_1 and st_{ℓ_1}; it uses pad_1 to recover $m_1 := \mathsf{pad}_1 \oplus c_1$. Later, to encrypt a second message m_2 of length ℓ_2, the sender will repeatedly run $\mathsf{GetBits}$ a total of ℓ_2 times, beginning at st_{ℓ_1}, to obtain $\mathsf{pad}_2 \stackrel{\text{def}}{=} y_{\ell_1+1}, \ldots, y_{\ell_1+\ell_2}$ and updated state $\mathsf{st}_{\ell_1+\ell_2}$, and then compute the ciphertext $c_2 := \mathsf{pad}_2 \oplus m_2$, and so on. This can continue indefinitely, allowing the parties to send an unlimited number of messages of arbitrary length. We remark that in this mode, the stream cipher does not need to use an IV.

This method of encrypting multiple messages requires the communicating parties to maintain synchronized state, explaining the terminology "synchronized mode." For that reason, the method is appropriate when two parties are communicating within a single "session," but it does not work well for sporadic communication or when a party might, over time, communicate from different devices. (It is relatively easy to keep copies of a fixed key in different locations; it is harder to maintain synchronized state across multiple locations.) Furthermore, if the parties ever get out of sync (e.g., because one of the transmissions between the parties is dropped), decryption will return an incorrect result. Resynchronization is possible, but adds additional overhead.

Unsynchronized mode. For stream ciphers whose Init function accepts an initialization vector as input, we can achieve *stateless* CPA-secure encryption for messages of arbitrary length. Here we modify G_∞ to accept three inputs: a seed s, an initialization vector IV, and a desired output length 1^ℓ. Now this algorithm first computes $\mathsf{st}_0 := \mathsf{Init}(s, IV)$ before repeatedly running $\mathsf{GetBits}$ a total of ℓ times. Encryption can then be carried out using a variant of Construction 3.30: the encryption of a message m using the key k is done by choosing a uniform initialization vector $IV \in \{0,1\}^n$ and computing the ciphertext $\langle IV, G_\infty(s, IV, 1^{|m|}) \oplus m\rangle$; decryption is performed in the natural way. (See Figure 3.4.) This scheme is CPA-secure if the stream cipher now has the stronger property that, for any polynomial ℓ, the function F defined by $F_k(IV) \stackrel{\text{def}}{=} G_\infty(k, IV, 1^\ell)$ is a pseudorandom function. (In fact, F need only be pseudorandom when evaluated on *uniform* inputs. Keyed functions with this weaker property are called *weak* pseudorandom functions.)

3.6.2 Block-Cipher Modes of Operation

We have already seen a construction of a CPA-secure encryption scheme based on pseudorandom functions/block ciphers. But Construction 3.30 (and its extension to arbitrary-length messages as discussed at the end of Section 3.4.2) have the drawback that the length of the ciphertext is *double* the length of the plaintext. Block-cipher modes of operation provide a way of encrypting arbitrary-length messages using shorter ciphertexts.

In this section, let F be a block cipher with block length n. We assume here that all messages m being encrypted have length a multiple of n, and write $m = m_1, m_2, \ldots, m_\ell$ where each $m_i \in \{0,1\}^n$ represents a block of the plaintext. Messages that are not a multiple of n can always be unambiguously padded to have length a multiple of n by appending a 1 followed by sufficiently many 0s, and so this assumption is without much loss of generality.

Several block-cipher modes of operation are known; we present four of the most common ones and discuss their security.

Electronic Code Book (ECB) mode. This is a naive mode of operation in which the ciphertext is obtained by direct application of the block cipher to each plaintext block. That is, $c := \langle F_k(m_1), F_k(m_2), \ldots, F_k(m_\ell)\rangle$; see

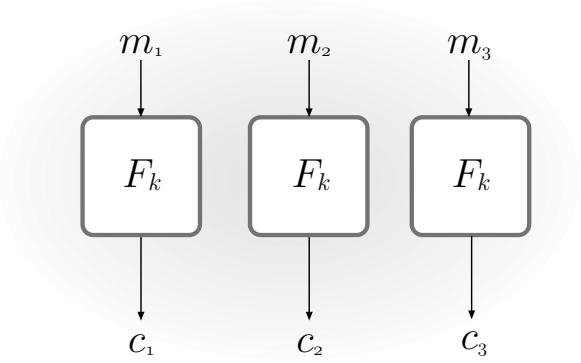


FIGURE 3.5: Electronic Code Book (ECB) mode.

Figure 3.5. Decryption is done in the obvious way, using the fact that F_k^{-1} is efficiently computable.

ECB mode is *deterministic* and therefore cannot be CPA-secure. Worse, ECB-mode encryption does not even have indistinguishable encryptions in the presence of an eavesdropper. This is because if a block is repeated in the plaintext, it will result in 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. This is not just a theoretical problem. Consider encrypting an image in which small groups of pixels correspond to a plaintext block. Encrypting using ECB mode may reveal a significant amount of information about patterns in the image, something that should not happen when using a secure encryption scheme. Figure 3.6 demonstrates this.

For these reasons, ECB mode should never be used. (We include it only because of its historical significance.)

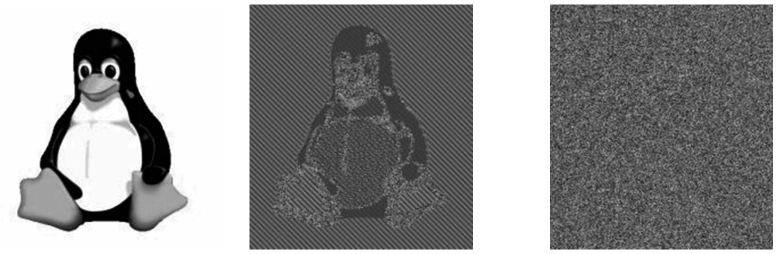


FIGURE 3.6: An illustration of the dangers of using ECB mode. The middle figure is an encryption of the image on the left using ECB mode; the figure on the right is an encryption of the same image using a secure mode. (Taken from `http://en.wikipedia.org` and derived from images created by Larry Ewing (`lewing@isc.tamu.edu`) using The GIMP.)

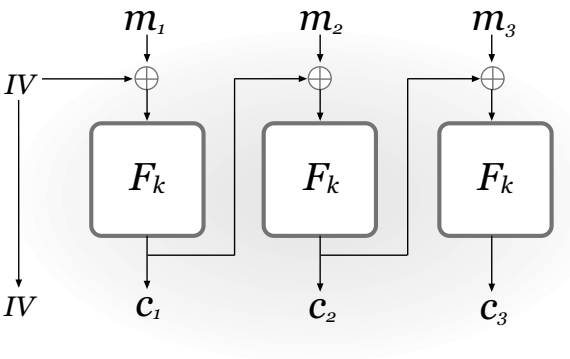


FIGURE 3.7: Cipher Block Chaining (CBC) mode.

Cipher Block Chaining (CBC) mode. To encrypt using this mode, a uniform initialization vector (IV) of length n is first chosen. Then, ciphertext blocks are generated by applying the block cipher to the XOR of the current plaintext block and the previous ciphertext block. That is, set $c_0 := IV$ and then, for $i = 1$ to ℓ, set $c_i := F_k(c_{i-1} \oplus m_i)$. The final ciphertext is $\langle c_0, c_1, \ldots, c_\ell \rangle$. (See Figure 3.7.) Decryption of a ciphertext $c_0, \ldots, c_\ell$ is done by computing $m_i := F_k^{-1}(c_i) \oplus c_{i-1}$ for $i = 1, \ldots, \ell$. We stress that the IV is included in the ciphertext; this is crucial so decryption can be carried out.

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

One may be tempted to think that it suffices to use a *distinct* IV (rather than a random IV) for every encryption, e.g., to first use $IV = 1$ and then increment the IV by one each time a message is encrypted. In Exercise 3.20, we ask you to show that this variant of CBC-mode encryption is not secure.

One might also consider a stateful variant of CBC-mode encryption—called *chained CBC mode*—in which the last block of the previous ciphertext is used as the IV when encrypting the next message. This reduces the bandwidth, as the IV need not be sent each time. See Figure 3.8, where an initial message m_1, m_2, m_3 is encrypted using a random IV, and then subsequently a second message m_4, m_5 is encrypted using c_3 as the IV. (In contrast, encryption using stateless CBC mode would generate a fresh IV when encrypting the second message.) Chained CBC mode is used in SSL 3.0 and TLS 1.0.

It may appear that chained CBC mode is as secure as CBC mode, since the chained-CBC encryption of m_1, m_2, m_3 followed by encryption of m_4, m_5 yields the same ciphertext blocks as CBC-mode encryption of the (single) message m_1, m_2, m_3, m_4, m_5. Nevertheless, chained CBC mode is vulnerable to a chosen-plaintext attack. The basis of the attack is that the adversary knows in

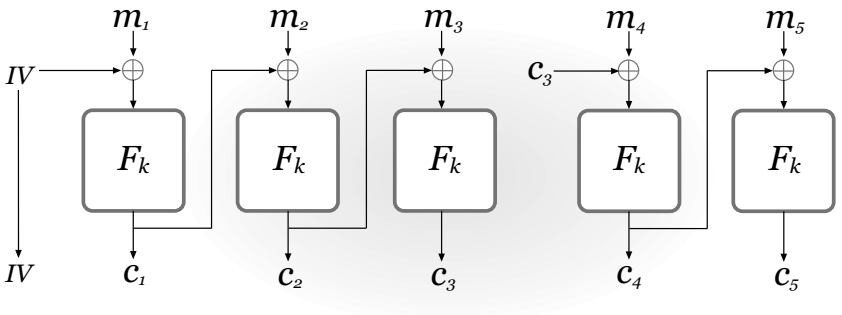


FIGURE 3.8: Chained CBC.

advance the "initialization vector" that will be used for the second encrypted message. We describe the attack informally, based on Figure 3.8. Assume the attacker knows that $m_1 \in \{m_1^0, m_1^1\}$, and observes the first ciphertext IV, c_1, c_2, c_3. The attacker then requests an encryption of a second message m_4, m_5 with $m_4 = IV \oplus m_1^0 \oplus c_3$, and observes a second ciphertext c_4, c_5. One can verify that $m_1 = m_1^0$ if and only if $c_4 = c_1$. This example should serve as a strong warning against making any modifications to cryptographic schemes, even if those modifications seem benign.

Output Feedback (OFB) mode. The third mode we present can be viewed as an unsynchronized stream-cipher mode, where the stream cipher is constructed in a specific way from the underlying block cipher. We describe the mode directly. First, a uniform $IV \in \{0,1\}^n$ is chosen. Then, a pseudorandom stream is generated from IV in the following way: Define $y_0 := IV$, and set the ith block y_i of the stream to be $y_i := F_k(y_{i-1})$. Each block of the plaintext is encrypted by XORing it with the appropriate block of the stream; that is, $c_i := y_i \oplus m_i$. (See Figure 3.9.) As in CBC mode, the IV is included as part of the ciphertext to enable decryption. However, in contrast to CBC mode, here it is not required that F be invertible. (In fact, it need

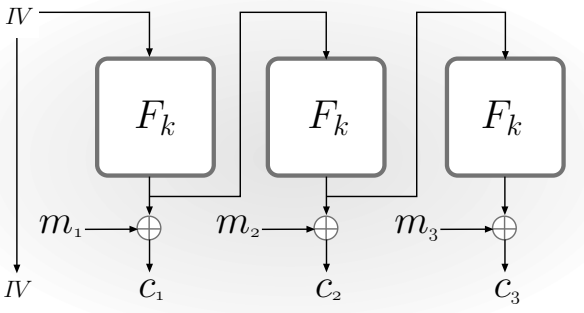


FIGURE 3.9: Output Feedback (OFB) mode.

not even be a permutation.) Furthermore, as in stream-cipher modes of operation, here it is not necessary for the plaintext length to be a multiple of the block length. Instead, the generated stream can be truncated to exactly the plaintext length. Another advantage of OFB mode is that its stateful variant (in which the final value y_ℓ used to encrypt some message is used as the IV for encrypting the next message) *is* secure. This stateful variant is equivalent to a *synchronized* stream-cipher mode, with the stream cipher constructed from the block cipher in the specific manner just described.

OFB mode can be shown to be CPA-secure if F is a pseudorandom function. Although both encryption and decryption must be carried out sequentially, this mode has the advantage relative to CBC mode 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 is possible to generate a pseudorandom stream ahead of time using preprocessing, after which point encryption of the plaintext (once it is known) is incredibly fast.

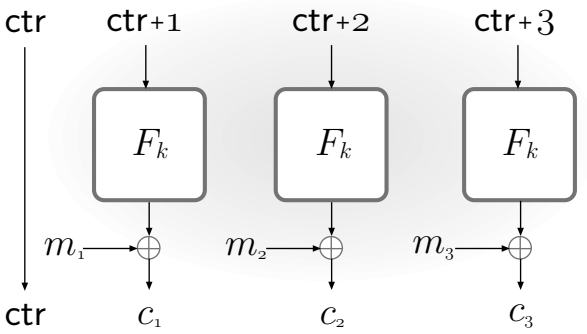


FIGURE 3.10: Counter (CTR) mode.

Counter (CTR) mode. Counter mode can also be viewed as an unsynchronized stream-cipher mode, where the stream cipher is constructed from the block cipher as in Construction 3.29. We give a self-contained description here. To encrypt using CTR mode, a uniform value $\mathsf{ctr} \in \{0,1\}^n$ is first chosen. Then, a pseudorandom stream is generated by computing $y_i := F_k(\mathsf{ctr} + i)$, where ctr and i are viewed as integers and addition is done modulo 2^n. The ith ciphertext block is $c_i := y_i \oplus m_i$, and the IV is again sent as part of the ciphertext; see Figure 3.10. Note again that decryption does not require F to be invertible, or even a permutation. As with OFB mode, another "stream-cipher" mode, the generated stream can be truncated to exactly the plaintext length, preprocessing can be used to generate the pseudorandom stream before the message is known, and the stateful variant of CTR mode is secure.

In contrast to all the secure modes discussed previously, CTR mode has the advantage that encryption and decryption can be fully *parallelized*, since all the blocks of the pseudorandom stream can be computed independently of each other. In contrast to OFB, it is also possible to decrypt the ith block of

the ciphertext using only a single evaluation of F. These features make CTR mode an attractive choice.

CTR mode is also fairly simple to analyze:

THEOREM 3.32 *If F is a pseudorandom function, then CTR mode is CPA-secure.*

PROOF We follow the same template as in the proof of Theorem 3.31: we first replace F with a random function and then analyze the resulting scheme.

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be the (stateless) CTR-mode encryption scheme, and let $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ be the encryption scheme that is identical to Π except that a truly random function is used in place of F_k. That is, $\widetilde{\mathsf{Gen}}(1^n)$ chooses a uniform function $f \in \mathsf{Func}_n$, and $\widetilde{\mathsf{Enc}}$ encrypts just like Enc except that f is used instead of F_k. (Once again, neither $\widetilde{\mathsf{Gen}}$ nor $\widetilde{\mathsf{Enc}}$ is efficient but this does not matter for the purposes of defining an experiment involving $\widetilde{\Pi}$.)

Fix an arbitrary PPT adversary $\mathcal{A}$, and let $q(n)$ be a polynomial upper-bound on the number of encryption-oracle queries made by $\mathcal{A}(1^n)$ as well as the maximum number of blocks in any such query and the maximum number of blocks in m_0 and m_1. As the first step of the proof, we claim that there is a negligible function negl such that

$$\left|\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(n) = 1\right] - \Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right]\right| \leq \mathsf{negl}(n). \tag{3.12}$$

This is proved by reduction in a way similar to the analogous step in the proof of Theorem 3.31, and is left as an exercise for the reader.

We next claim that

$$\Pr\left[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n) = 1\right] < \frac{1}{2} + \frac{2q(n)^2}{2^n}. \tag{3.13}$$

Combined with Equation (3.12) this means that

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

Since q is polynomial, $\frac{2q(n)^2}{2^n}$ is negligible and this completes the proof.

We now prove Equation (3.13). Fix some value n for the security parameter. Let $\ell^* \leq q(n)$ denote the length (in blocks) of the messages m_0, m_1 output by $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A},\widetilde{\Pi}}(n)$, and let ctr^* denote the initial value used when generating the challenge ciphertext. Similarly, let $\ell_i \leq q(n)$ be the length (in blocks) of the ith encryption-oracle query made by $\mathcal{A}$, and let ctr_i denote the initial value used when answering this query. When the ith encryption-oracle query is answered, f is applied to the values $\mathsf{ctr}_i + 1, \ldots, \mathsf{ctr}_i + \ell_i$. When the challenge ciphertext is encrypted, f is applied to $\mathsf{ctr}^*+1, \ldots, \mathsf{ctr}^*+\ell^*$, and the ith ciphertext block is computed by XORing $f(\mathsf{ctr}^* + i)$ with the ith message block. There are two cases:

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

2. *There exist $i, j, j^* \geq 1$ (with $j \leq \ell_i$ and $j^* \leq \ell^*$) for which $\mathsf{ctr}_i + j = \mathsf{ctr}^* + j^*$:* We denote this event by $\mathsf{Overlap}$. In this case $\mathcal{A}$ may potentially determine which of its messages was encrypted to give the challenge ciphertext, since $\mathcal{A}$ can deduce the value of $f(\mathsf{ctr}_i + j) = f(\mathsf{ctr}^* + j^*)$ from the answer to its ith oracle query.

Let us analyze the probability that $\mathsf{Overlap}$ occurs. The probability is maximized if ℓ^* and each ℓ_i are as large as possible, so assume $\ell^* = \ell_i = q(n)$ for all i. Let $\mathsf{Overlap}_i$ denote the event that the sequence $\mathsf{ctr}_i + 1, \ldots, \mathsf{ctr}_i + q(n)$ overlaps the sequence $\mathsf{ctr}^* + 1, \ldots, \mathsf{ctr}^* + q(n)$; then $\mathsf{Overlap}$ is the event that $\mathsf{Overlap}_i$ occurs for some i. Since there are at most $q(n)$ oracle queries, a union bound (cf. Proposition A.7) gives

$$\Pr[\mathsf{Overlap}] \leq \sum_{i=1}^{q(n)} \Pr[\mathsf{Overlap}_i]. \tag{3.14}$$

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

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

Since there are $2q(n) - 1$ values of ctr_i for which $\mathsf{Overlap}_i$ occurs, and ctr_i is chosen uniformly from $\{0,1\}^n$, we see that

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

Combined with Equation (3.14), this gives $\Pr[\mathsf{Overlap}] < 2q(n)^2/2^n$.

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

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

This proves Equation (3.13) and completes the proof. ■

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

With regard to the behavior of different modes in the presence of "benign" (i.e., non-adversarial) transmission errors, see Exercises 3.21 and 3.22. We remark, however, that in general such errors can be addressed using standard techniques (e.g., error correction or re-transmission).

Block length and concrete security. CBC, OFB, and CTR modes all use a random IV. This has the effect of randomizing the encryption process, and ensures that (with high probability) the underlying block cipher is always evaluated on *fresh* (i.e., new) inputs. This is important because, as we have seen in the proofs of Theorem 3.31 and Theorem 3.32, if an input to the block cipher is used more than once then security can be violated.

The block length of a block cipher thus has a significant impact on the concrete security of encryption schemes based on that cipher. Consider, e.g., CTR mode using a block cipher F with block length ℓ. The IV is then a uniform ℓ-bit string, and we expect an IV to repeat after encrypting about $2^{\ell/2}$ messages (see Appendix A.4). If ℓ is too short, then—even if F is secure as a pseudorandom permutation—the resulting concrete-security bound will be too weak for practical applications. Concretely, if $\ell = 64$ as is the case for DES (a block cipher we will study in Chapter 6), then after $2^{32} \approx 4{,}300{,}000{,}000$ encryptions—or roughly 34 gigabytes of plaintext—a repeated IV is expected to occur. Although this may seem like a lot of data, it is smaller than the capacity of modern hard drives.

IV misuse. In our description and discussion of the various (secure) modes, we have assumed a random IV is chosen each time a message encrypted. What if this assumption goes wrong due, e.g., to poor randomness generation or a mistaken implementation? Certainly we can no longer guarantee security in the sense of Definition 3.22. From a practical point of view, though, the "stream-cipher modes" (OFB and CTR) are much worse than CBC mode. If an IV repeats when using the former, an attacker can XOR the two resulting ciphertexts and learn a lot of information about the *entire* contents of both encrypted messages (as we have seen previously in the context of the one-time pad if the key is re-used). With CBC mode, however, it is likely that after only a few blocks the inputs to the block cipher will "diverge" and the attacker will be unable to learn any information beyond the first few message blocks.

One way to address potential IV misuse is to use stateful encryption, as discussed in the context of OFB and CTR modes. If stateful encryption is not possible, and there is concern about potential IV misuse, then CBC mode is recommended for the reasons described above.

3.7 Chosen-Ciphertext Attacks

3.7.1 Defining CCA-Security

Until now we have defined security against two types of attacks: passive eavesdropping and chosen-plaintext attacks. A *chosen-ciphertext attack* is even more powerful. In a chosen-ciphertext attack, the adversary has the ability not only to obtain encryptions of messages of its choice (as in a chosen-plaintext attack), but also to obtain the *decryption* of ciphertexts of its choice (with one exception discussed later). Formally, we give the adversary access to a *decryption oracle* in addition to an encryption oracle. We present the formal definition and defer further discussion.

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

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

1. *A key k is generated by running* $\mathsf{Gen}(1^n)$.
2. *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 uniform bit $b \in \{0,1\}$ 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. If the output of the experiment is* 1, *we say that $\mathcal{A}$* succeeds.

DEFINITION 3.33 *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 is 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 randomness used in the experiment.

For completeness, we remark that the natural analogue of Theorem 3.24 holds for CCA-security as well. (Namely, if a scheme has indistinguishable encryptions under a chosen-ciphertext attack then it has indistinguishable *multiple* encryptions under a chosen-ciphertext attack, defined appropriately.)

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 the definition.

At this point you may be wondering if chosen-ciphertext attacks realistically model any real-world attack. As in the case of chosen-plaintext attacks, 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. For example:

1. In the Midway example from Section 3.4.2, 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 sending encrypted messages to their bank. An adversary may be able to send ciphertexts on behalf of that user; the bank will decrypt those ciphertexts and the adversary may learn something about the result. For example, if a ciphertext corresponds to an *ill-formed* plaintext (e.g., an unintelligible message, or simply one that is not formatted correctly), the adversary may be able to deduce this from the bank's reaction (i.e., the pattern of subsequent communication). A practical attack of this type is presented in Section 3.7.2 below.

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

Insecurity of the schemes we have studied. None of the encryption schemes we have seen thus far is CCA-secure. We demonstrate this for Construction 3.30, where encryption is computed as $\mathsf{Enc}_k(m) = \langle r, F_k(r) \oplus m \rangle$. Consider an adversary $\mathcal{A}$ running in the CCA indistinguishability experiment who chooses $m_0 = 0^n$ and $m_1 = 1^n$. Then, upon receiving a ciphertext $c = \langle r, s \rangle$, the adversary 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 that CCA-security is quite stringent. Any encryption scheme that allows ciphertexts to be "manipulated" in any controlled way cannot be CCA-secure. Thus, CCA-security 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 invalid ciphertext or one

that decrypts to a plaintext having no relation to the original one. This is a very useful property for schemes used in complex cryptographic protocols.

Constructing a CCA-secure encryption scheme. We show how to construct a CCA-secure encryption scheme in Section 4.5.4. The construction is presented there because it uses tools that we develop in Chapter 4.

3.7.2 Padding-Oracle Attacks

The chosen-ciphertext attack on Construction 3.30 described in the previous section is a bit contrived, since it assumes the attacker can obtain the complete decryption of a modified ciphertext. While this sort of attack is allowed under Definition 3.33, it is not clear that it represents a serious concern in practice. Here we show a chosen-ciphertext attack on a natural and widely used encryption scheme; moreover, the attack only requires the ability of an attacker to determine whether or not a modified ciphertext decrypts correctly. Such information is frequently very easy to obtain since, for example, a server might request a retransmission or terminate a session if it ever receives a ciphertext that does not decrypt correctly, and either of these events would generate a noticeable change in the observed traffic. The attack has been shown to work in practice on various deployed protocols; we give one concrete example at the end of this section.

As mentioned previously, when using CBC mode the length of the plaintext must be a multiple of the block length; if a plaintext does not satisfy this property, it must be padded before being encrypted. We refer to the original plaintext as the *message*, and the result after padding as *encoded data*. The padding scheme we use must allow the receiver to unambiguously determine from the encoded data where the message ends. One popular and standardized approach is to use PKCS #5 padding. Assume the original message has an integral number of bytes, and let L denote the block length (in bytes) of the block cipher being used. Let b denote the number of bytes that need to be appended to the message in order to make the total length of the resulting encoded data a multiple of the block length. Here, b is an integer between 1 and L, inclusive. (We cannot have $b = 0$ since this would lead to ambiguous padding. Thus, if the message length is already a multiple of the block length, L bytes of padding are appended.) Then we append to the message the string containing the integer b (represented in 1 byte, or 2 hexadecimal digits) repeated b times. That is, if 1 byte of padding is needed then the string 0x01 (written in hexadecimal) is appended; if 4 bytes of padding are needed then the hexadecimal string 0x04040404 is appended; etc. The encoded data is then encrypted using regular CBC-mode encryption.

When decrypting, the receiver first applies CBC-mode decryption as usual to recover the encoded data, and then checks that the encoded data is correctly padded. (This is easily done: simply read the value b of the final byte and then verify that the final b bytes of the result all have value b.) If so,

then the padding is stripped off and the original message returned. Otherwise, the standard procedure is to return a "bad padding" error (e.g., in Java the standard exception is called `javax.crypto.BadPaddingException`). The presence of such an error message provides an adversary with a *partial* decryption oracle. That is, an adversary can send any ciphertext to the server and learn (based on whether a "bad padding" error is returned or not) whether the underlying encoded data is padded in the correct format. Although this may seem like meaningless information, we show that it enables an adversary to completely recover the original message for any ciphertext of its choice.

We describe the attack on a 3-block ciphertext for simplicity. Let IV, c_1, c_2 be a ciphertext observed by the attacker, and let m_1, m_2 be the underlying encoded data (unknown to the attacker) which corresponds to the padded message, as discussed above. (Each block is L bytes long.) Note that

$$m_2 = F_k^{-1}(c_2) \oplus c_1, \tag{3.15}$$

where k is the key (which is, of course, not known to the attacker) being used by the honest parties. The second block m_2 ends in $\underbrace{0\text{x}b \cdots 0\text{x}b}_{b \text{ times}}$, where we let $0\text{x}b$ denote the 1-byte representation of the integer b. The key property used in the attack is that certain changes to the ciphertext yield predictable changes in the underlying encoded data after CBC-mode decryption. Specifically, let c_1' be identical to c_1 except for a modification in the final byte. Consider decryption of the modified ciphertext IV, c_1', c_2. This will result in encoded data m_1', m_2' where $m_2' = F_k^{-1}(c_2) \oplus c_1'$. Comparing to Equation (3.15) we see that m_2' will be identical to m_2 except for a modification in the final byte. (The value of m_1' is unpredictable, but this will not adversely affect the attack.) Similarly, if c_1' is the same as c_1 except for a change in its ith byte, then decryption of IV, c_1', c_2 will result in m_1', m_2' where m_2' is the same as m_2 except for a change in its ith byte. More generally, if $c_1' = c_1 \oplus \Delta$ for any string Δ, then decryption of IV, c_1', c_2 yields m_1', m_2' where $m_2' = m_2 \oplus \Delta$. The upshot is that the attacker can exercise significant control over the final block of the encoded data.

As a warmup, let us see how the adversary can exploit this to learn b, the amount of padding. (This reveals the length of the original message.) Recall that upon decryption, the receiver looks at the value b of the final byte of the second block of the encoded data, and then verifies that the final b bytes all have the same value. The attacker begins by modifying the first byte of c_1 and sending the resulting ciphertext IV, c_1', c_2 to the receiver. If decryption fails (i.e., the receiver returns an error) then it must be the case that the receiver is checking all L bytes of m_2', and therefore $b = L$! Otherwise, the attacker learns that $b < L$, and it can then repeat the process with the second byte, and so on. The left-most modified byte for which decryption fails reveals exactly the left-most byte being checked by the receiver, and so reveals exactly b.

With b known, the attacker can proceed to learn the bytes of the message one-by-one. We illustrate the idea for the final byte of the message, which we

denote by B. The attacker knows that m_2 ends in $0\text{x}B0\text{x}b \cdots 0\text{x}b$ (with $0\text{x}b$ repeated b times) and wishes to learn B. Define

$$\Delta_i \stackrel{\text{def}}{=} 0\text{x}00 \cdots 0\text{x}00\ 0\text{x}i\ \overbrace{0\text{x}(b+1) \cdots 0\text{x}(b+1)}^{b \text{ times}}$$
$$\oplus\ 0\text{x}00 \cdots 0\text{x}00\ 0\text{x}00\ \overbrace{0\text{x}b \cdots 0\text{x}b}^{b \text{ times}}$$

for $0 \leq i < 2^8$; i.e., the final $b+1$ byes of Δ_i contain the integer i (represented in hexadecimal) followed by the value $(b+1) \oplus b$ (in hexadecimal) repeated b times. If the attacker submits the ciphertext $IV, c_1 \oplus \Delta_i, c_2$ to the receiver then, after CBC-mode decryption, the resulting encoded data will equal $0\text{x}(B \oplus i)0\text{x}(b+1) \cdots 0\text{x}(b+1)$ (with $0\text{x}(b+1)$ repeated b times). Decryption will fail unless $0\text{x}(B \oplus i) = 0\text{x}(b+1)$. The attacker simply needs to try at most 2^8 values $\Delta_0, \ldots, \Delta_{2^8-1}$ until decryption succeeds for some Δ_i, at which point it can deduce that $B = 0\text{x}(b+1) \oplus i$. We leave a full description of how to extend this attack so as to learn the entire plaintext—and not just the final block—as an exercise.

A padding-oracle attack on CAPTCHAs. A CAPTCHA is a distorted image of, say, an English word that is easy for humans to read, but hard for a computer to process. CAPTCHAs are used in order to ensure that a human user—and not some automated software—is interacting with a webpage. CAPTCHAs are used, e.g., by online email services to ensure that humans open accounts; this is important to prevent spammers from automatically opening thousands of accounts and using them to send spam.

One way that CAPTCHAs can be configured is as a separate service run on an independent server. In order to see how this works, we denote a webserver by $\mathcal{S}_W$, the CAPTCHA server by $\mathcal{S}_C$, and the user by $\mathcal{U}$. When the user $\mathcal{U}$ loads a webpage served by $\mathcal{S}_W$, the following events occur: $\mathcal{S}_W$ encrypts a random English word w using a key k that was initially shared between $\mathcal{S}_W$ and $\mathcal{S}_C$, and sends the resulting ciphertext to the user (along with the webpage). $\mathcal{U}$ forwards the ciphertext to $\mathcal{S}_C$, who decrypts it, obtains w, and renders a distorted image of w to $\mathcal{U}$. Finally, $\mathcal{U}$ sends the word w back to $\mathcal{S}_W$ for verification. The interesting observation here is that $\mathcal{S}_C$ decrypts any ciphertext it receives from $\mathcal{U}$ and will issue a "bad padding" error message if decryption fails as described earlier. This presents $\mathcal{U}$ with an opportunity to carry out a padding-oracle attack as described above, and thus to solve the CAPTCHA (i.e., to determine w) automatically *without any human involvement*, rendering the CAPTCHA ineffective. While ad hoc measures can be used (e.g., having $\mathcal{S}_C$ return a random image instead of a decryption error), what is really needed is to use an encryption scheme that is CCA-secure.

References and Additional Reading

The modern computational approach to cryptography was initiated in a groundbreaking paper by Goldwasser and Micali [80]. 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 10 and 11). That paper also proposed the notion of indistinguishability (i.e., Definition 3.8), and showed that it implies semantic security. The converse was shown in [125]. The reader is referred to [76] for further discussion of semantic security.

Blum and Micali [37] introduced the notion of pseudorandom generators and proved their existence based on a specific number-theoretic assumption. In the same work, Blum and Micali also pointed out the connection between pseudorandom generators and private-key encryption as in Construction 3.17. The definition of pseudorandom generators given by Blum and Micali is different from the definition we use in this book (Definition 3.14); the latter definition originates in the work of Yao [179], who showed equivalence of the two formulations. Yao also showed constructions of pseudorandom generators based on general assumptions; we will explore this result in Chapter 7.

Formal definitions of security against chosen-plaintext attacks were given by Luby [115] and Bellare et al. [15]. Chosen-ciphertext attacks (in the context of public-key encryption) were first formally defined by Naor and Yung [129] and Rackoff and Simon [147], and were considered also in [61] and [15]. See [101] for other notions of security for private-key encryption.

Pseudorandom functions were defined and constructed by Goldreich et al. [78], and their application to encryption was demonstrated in subsequent work by the same authors [77]. Pseudorandom permutations and strong pseudorandom permutations were studied by Luby and Rackoff [116]. A proof of Proposition 3.27 (the "switching lemma") can be found in [24]. Block ciphers had been used for many years before they began to be studied in the theoretical sense initiated by the above works. Practical stream ciphers and block ciphers are studied in detail in Chapter 6.

The ECB, CBC, and OFB modes of operation (as well as CFB, a mode of operation not covered here) were standardized along with the DES block cipher [130]. CTR mode was standardized by NIST in 2001. CBC and CTR mode were proven CPA-secure in [15]. For more recent modes of operation, see `http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html`.

The attack on chained CBC was first described by Rogaway (unpublished), and was used to attack SSL/TLS in the so-called "BEAST attack" by Rizzo and Duong. The padding-oracle attack we describe here originated in the work of Vaudenay [171].

Exercises

3.1 Prove Proposition 3.6.

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

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

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

3.4 Prove the equivalence of Definition 3.8 and Definition 3.9.

3.5 Let $|G(s)| = \ell(|s|)$ for some ℓ. Consider the following experiment:

> **The PRG indistinguishability experiment $\mathsf{PRG}_{\mathcal{A},G}(n)$:**
>
> *(a) A uniform bit $b \in \{0,1\}$ is chosen. If $b = 0$ then choose a uniform $r \in \{0,1\}^{\ell(n)}$; if $b = 1$ then choose a uniform $s \in \{0,1\}^n$ and set $r := G(s)$.*
>
> *(b) The adversary $\mathcal{A}$ is given r, and outputs a bit b'.*
>
> *(c) The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.*

Provide a definition of a pseudorandom generator based on this experiment, and prove that your definition is equivalent to Definition 3.14. (That is, show that G satisfies your definition if and only if it satisfies Definition 3.14.)

3.6 Let G be a pseudorandom generator with expansion factor $\ell(n) > 2n$. In each of the following cases, say whether G' is necessarily a pseudorandom generator. If yes, give a proof; if not, show a counterexample.

(a) Define $G'(s) \stackrel{\text{def}}{=} G(s_1 \cdots s_{\lfloor n/2 \rfloor})$, where $s = s_1 \cdots s_n$.

(b) Define $G'(s) \stackrel{\text{def}}{=} G\left(0^{|s|} \| s\right)$.

(c) Define $G'(s) \stackrel{\text{def}}{=} G(s) \,\|\, G(s+1)$.

3.7 Prove the converse of Theorem 3.18. Namely, show that if G is not a pseudorandom generator then Construction 3.17 does not have indistinguishable encryptions in the presence of an eavesdropper.

3.8 (a) Define a notion of indistinguishability for the encryption of multiple *distinct* messages, in which a scheme need not hide whether the same message is encrypted twice.

(b) Show that Construction 3.17 does not satisfy your definition.

(c) Give a construction of a *deterministic* (stateless) encryption scheme that satisfies your definition.

3.9 Prove *unconditionally* the existence of a pseudorandom function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ with $\ell_{key}(n) = n$ and $\ell_{in}(n) = O(\log n)$.

Hint: Implement a uniform function with logarithmic input length.

3.10 Let F be a length-preserving pseudorandom function. For the following constructions of a keyed function $F' : \{0,1\}^n \times \{0,1\}^{n-1} \to \{0,1\}^{2n}$, state whether F' is a pseudorandom function. If yes, prove it; if not, show an attack.

(a) $F'_k(x) \stackrel{\text{def}}{=} F_k(0\|x) \,\|\, F_k(1\|x)$.

(b) $F'_k(x) \stackrel{\text{def}}{=} F_k(0\|x) \,\|\, F_k(x\|1)$.

3.11 Assuming the existence of pseudorandom functions, prove that there is an encryption scheme that has indistinguishable multiple encryptions in the presence of an eavesdropper (i.e., satisfies Definition 3.19), but is not CPA-secure (i.e., does not satisfy Definition 3.22).

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

3.12 Let F be a keyed function and consider the following experiment:

The PRF indistinguishability experiment $\mathsf{PRF}_{\mathcal{A},F}(n)$:

(a) A uniform bit $b \in \{0,1\}$ is chosen. If $b = 1$ then choose uniform $k \in \{0,1\}^n$.

(b) $\mathcal{A}$ is given 1^n for input. If $b = 0$ then $\mathcal{A}$ is given access to a uniform function $f \in \mathsf{Func}_n$. If $b = 1$ then $\mathcal{A}$ is instead given access to $F_k(\cdot)$.

(c) $\mathcal{A}$ outputs a bit b'.

(d) The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

Define pseudorandom functions using this experiment, and prove that your definition is equivalent to Definition 3.25.

3.13 Consider the following keyed function F: For security parameter n, the key is an $n \times n$ boolean matrix A and an n-bit boolean vector b. Define $F_{A,b} : \{0,1\}^n \to \{0,1\}^n$ by $F_{A,b}(x) \stackrel{\text{def}}{=} Ax + b$, where all operations are done modulo 2. Show that F is not a pseudorandom function.

3.14 Prove that if F is a length-preserving pseudorandom function, then $G(s) \stackrel{\text{def}}{=} F_s(1)\|F_s(2)\|\cdots\|F_s(\ell)$ is a pseudorandom generator with expansion factor $\ell \cdot n$.

3.15 Define a notion of perfect secrecy under a chosen-plaintext attack by adapting Definition 3.22. Show that the definition cannot be achieved.

3.16 Prove Proposition 3.27.

Hint: Use the results of Appendix A.4.

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

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

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

Show how to decrypt, and prove that this scheme is CPA-secure for messages of length $n/2$. (If you are looking for a real challenge, prove that this scheme is CCA-secure if F is a *strong* pseudorandom permutation.)

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

(a) To encrypt $m \in \{0,1\}^{n+1}$, choose uniform $r \in \{0,1\}^n$ and output the ciphertext $\langle r, G(r) \oplus m\rangle$.

(b) To encrypt $m \in \{0,1\}^n$, output the ciphertext $m \oplus F_k(0^n)$.

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

3.20 Consider a stateful 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.

3.21 What is the effect of a single-bit error in the ciphertext when using the CBC, OFB, and CTR modes of operation?

3.22 What is the effect of a dropped ciphertext block (e.g., if the trabsmitted ciphertext $c_1, c_2, c_3, \ldots$ is received as $c_1, c_3, \ldots$) when using the CBC, OFB, and CTR modes of operation?

3.23 Say CBC-mode encryption is used with a block cipher having a 256-bit key and 128-bit block length to encrypt a 1024-bit message. What is the length of the resulting ciphertext?

3.24 Give the details of the proof by reduction for Equation (3.12).

3.25 Let F be a pseudorandom function such that for $k \in \{0,1\}^n$ the function F_k maps $\ell_{in}(n)$-bit inputs to $\ell_{out}(n)$-bit outputs.

(a) Consider implementing CTR-mode encryption using F. For which functions ℓ_{in}, ℓ_{out} is the resulting encryption scheme CPA-secure?

(b) Consider implementing CTR-mode encryption using F, but only for *fixed-length* messages of length $\ell(n)$ (which is an integer multiple of $\ell_{out}(n)$). For which $\ell_{in}, \ell_{out}, \ell$ does the scheme have indistinguishable encryptions in the presence of an eavesdropper?

3.26 For any function $g : \{0,1\}^n \to \{0,1\}^n$, define $g^{\$}(\cdot)$ to be a *probabilistic* oracle that, on input 1^n, chooses uniform $r \in \{0,1\}^n$ and returns $\langle r, g(r)\rangle$. A keyed function F is a *weak pseudorandom function* if for all PPT algorithms D, there exists a negligible function negl such that:

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

where $k \in \{0,1\}^n$ and $f \in \mathsf{Func}_n$ are chosen uniformly.

(a) Prove that if F is pseudorandom then it is weakly pseudorandom.

(b) Let F' be a pseudorandom function, and define

$$F_k(x) \stackrel{\text{def}}{=} \begin{cases} F'_k(x) & \text{if } x \text{ is even} \\ F'_k(x+1) & \text{if } x \text{ is odd.} \end{cases}$$

Prove that F is weakly pseudorandom, but *not* pseudorandom.

(c) Is CTR-mode encryption using a weak pseudorandom function necessarily CPA-secure? Does it necessarily have indistinguishable encryptions in the presence of an eavesdropper? Prove your answers.

(d) Prove that Construction 3.30 is CPA-secure if F is a weak pseudorandom function.

3.27 Let F be a pseudorandom permutation. Consider the mode of operation in which a uniform value $\mathsf{ctr} \in \{0,1\}^n$ is chosen, and the ith ciphertext block c_i is computed as $c_i := F_k(\mathsf{ctr} + i + m_i)$. Show that this scheme does not have indistinguishable encryptions in the presence of an eavesdropper.

3.28 Show that the CBC, OFB, and CTR modes of operation do not yield CCA-secure encryption schemes (regardless of F).

3.29 Let $\Pi_1 = (\mathsf{Enc}_1, \mathsf{Dec}_1)$ and $\Pi_2 = (\mathsf{Enc}_2, \mathsf{Dec}_2)$ be two encryption schemes for which it is known that at least one is CPA-secure (but you don't know which one). 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. Provide a full proof of your solution.

> **Hint:** Generate two plaintext messages from the original plaintext so that knowledge of either one reveals nothing about the original plaintext, but knowledge of both enables the original plaintext to be computed.

3.30 Write pseudocode for obtaining the entire plaintext via a padding-oracle attack on CBC-mode encryption using PKCS #5 padding, as described in the text.

3.31 Describe a padding-oracle attack on CTR-mode encryption (assuming PKCS #5 padding is used to pad messages to a multiple of the block length before encrypting).

Chapter 4

Message Authentication Codes

4.1 Message Integrity

4.1.1 Secrecy vs. Integrity

One of the most basic goals of cryptography is to enable parties to communicate over an *open communication channel* in a secure way. But what does "secure communication" entail? In Chapter 3 we showed that it is possible to obtain *secret* 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 secrecy. In many cases, it is of equal or greater importance to guarantee *message integrity* (or *message authentication*) in the sense that each party should be able to identify when a message it receives was sent by the party claiming to send it, and was not modified in transit. We look at two canonical examples.

Consider the case of a user communicating with their bank over the Internet. When the bank receives a request to transfer $1,000 from the user's account to the account of some other user X, the bank has to consider the following:

1. Is the request authentic? That is, did the user in question really issue this request, or was the request issued by an adversary (perhaps X itself) who is impersonating the legitimate user?

2. Assuming a transfer request was issued by the legitimate user, are the details of the request as received exactly those intended by the legitimate user? Or was, e.g., the transfer amount modified?

Note that standard error-correction techniques do not suffice for the second concern. Error-correcting codes are only intended to detect and recover from "random" errors that affect only a small portion of the transmission, but they do nothing to protect against a malicious adversary who can choose exactly where to introduce an arbitrary number of errors.

A second scenario where the need for message integrity arises in practice is with regard to web cookies. The HTTP protocol used for web traffic is *stateless*, and so when a client and server communicate in some session (e.g., when

a user [client] shops at a merchant's [server's] website), any state generated as part of that session (e.g., the contents of the user's shopping cart) is often placed in a "cookie" that is stored by the client and sent from the client to the server as part of each message the client sends. Assume that the cookie stored by some user includes the items in the user's shopping cart along with a price for each item, as might be done if the merchant offers different prices to different users (reflecting, e.g., discounts and promotions, or user-specific pricing). It should be infeasible here for the user to modify the cookie that it stores so as to alter the prices of the items in its cart. The merchant thus needs a technique to ensure the integrity of the cookie that it stores at the user. Note that the contents of the cookie (namely, the items and their prices) are not secret and, in fact, must be known by the user. The problem here is thus purely one of integrity.

In general, one cannot assume the integrity of communication without taking specific measures to ensure it. Indeed, any unprotected online purchase order, online banking operation, email, or SMS message cannot, in general, be trusted to have originated from the claimed source and to have been unmodified in transit. Unfortunately, people are in general trusting and thus information like the caller-ID or an email return address are taken to be "proofs of origin" in many cases, even though they are relatively easy to forge. This leaves the door open to potentially damaging attacks.

In this chapter we will show how to achieve message integrity by using cryptographic techniques to prevent the *undetected* tampering of messages sent over an open communication channel. Note that we cannot hope to prevent adversarial tampering of messages altogether, as that can only be defended against at the physical level. Instead, what we will guarantee is that any such tampering will be detected by the honest parties.

4.1.2 Encryption vs. Message Authentication

Just as the goals of secrecy and message integrity are different, so are the techniques and tools for achieving them. Unfortunately, secrecy and integrity are often confused and unnecessarily intertwined, so let us be clear up front: encryption does *not* (in general) provide any integrity, and encryption should never be used with the intent of achieving message authentication unless it is specifically designed with that purpose in mind (something we will return to in Section 4.5).

One might mistakenly think that encryption solves the problem of message authentication. (In fact, this is a common error.) This is due to the fuzzy, and incorrect, reasoning that since a ciphertext completely hides the contents of the message, an adversary cannot possibly modify an encrypted message in any meaningful way. Despite its intuitive appeal, this reasoning is completely false. We illustrate this point by showing that all the encryption schemes we have seen thus far do not provide message integrity.

Encryption using stream ciphers. Consider the simple encryption scheme in which $\mathsf{Enc}_k(m)$ computes the ciphertext $c := G(k) \oplus m$, where G is a pseudorandom generator. Ciphertexts in this case are very easy to manipulate: flipping any bit in the ciphertext c results in the same bit being flipped in the message that is recovered upon decryption. Thus, given a ciphertext c that encrypts a (possibly unknown) message m, it is possible to generate a modified ciphertext c' such that $m' := \mathsf{Dec}_k(c')$ is the same as m but with one (or more) of the bits flipped. This simple attack can have severe consequences. As an example, consider the case of a user encrypting some dollar amount he wants to transfer from his bank account, where the amount is represented in binary. Flipping the least significant bit has the effect of changing this amount by only \$1, but flipping the 11th least significant bit changes the amount by more than \$1,000! (Interestingly, the adversary in this example does not necessarily learn whether it is increasing or decreasing the initial amount, i.e., whether it is flipping a 0 to a 1 or vice versa. But if the adversary has some partial knowledge about the amount—say, that it is less than \$1,000 to begin with—then the modifications it introduces can have a predictable effect.) We stress that this attack does not contradict the *secrecy* of the encryption scheme (in the sense of Definition 3.8). In fact, the exact same attack applies to the one-time pad encryption scheme, showing that even perfect secrecy is not sufficient to ensure the most basic level of message integrity.

Encryption using block ciphers. The attack described above utilizes the fact that flipping a single bit in a ciphertext keeps the underlying plaintext unchanged except for the corresponding bit (which is also flipped). The same attack applies to the OFB- and CTR-mode encryption schemes, which also encrypt messages by XORing them with a pseudorandom stream (albeit one that changes each time a message is encrypted). We thus see that even using CPA-secure encryption is not enough to prevent message tampering.

One may hope that attacking ECB- or CBC-mode encryption would be more difficult since decryption in these cases involves inverting a (strong) pseudorandom permutation F, and we expect that $F_k^{-1}(x)$ and $F_k^{-1}(x')$ will be completely uncorrelated even if x and x' differ in only a single bit. (Of course, ECB mode does not even guarantee the most basic notion of secrecy, but that does not matter for the present discussion.) Nevertheless, single-bit modifications of a ciphertext still cause partially predictable changes in the plaintext. For example, when using ECB mode, flipping a bit in the ith block of the ciphertext affects *only* the ith block of the plaintext—all other blocks remain unchanged. Although the effect on the ith block of the plaintext may be impossible to predict, changing that one block (while leaving everything else unchanged) may represent a harmful attack. Moreover, the order of plaintext blocks can be changed (without garbling any block) by simply changing the order of the corresponding ciphertext blocks, and the message can be truncated by just dropping ciphertext blocks.

Similarly, when using CBC mode, flipping the jth bit of the IV changes only

the jth bit of the first message block m_1 (since $m_1 := F_k^{-1}(c_1) \oplus IV'$, where IV' is the modified IV); all plaintext blocks other than the first remain unchanged (since the ith block of the plaintext is computed as $m_i := F_k^{-1}(c_i) \oplus c_{i-1}$, and blocks c_i and c_{i-1} have not been modified). Therefore, the first block of a CBC-encrypted message can be changed arbitrarily. This a serious concern in practice since the first block often contains important header information.

Finally, note that all encryption schemes we have seen thus far have the property that *every* ciphertext (perhaps satisfying some length constraint) corresponds to *some* valid message. So it is trivial for an adversary to "spoof" a message on behalf of one of the communicating parties—by sending some arbitrary ciphertext—even if the adversary has no idea what the underlying message will be. As we will see when we formally define authenticated encryption in Section 4.5, even an attack of this sort should be ruled out.

4.2 Message Authentication Codes – Definitions

We have seen that, in general, encryption does not solve the problem of message integrity. Rather, an additional mechanism is needed that will enable the communicating parties to know whether or not a message was tampered with. The right tool for this task is a *message authentication code* (MAC).

The aim of a message authentication code is to prevent an adversary from modifying a message sent by one party to another, or from injecting a new message, without the receiver detecting that the message did not originate from the intended party. As in the case of encryption, this is only possible if the communicating parties share some secret that the adversary does not know (otherwise nothing can prevent an adversary from impersonating the party sending the message). Here, we will continue to consider the private-key setting where the parties share the same secret key.[1]

The Syntax of a Message Authentication Code

Before formally defining security of a message authentication code, we first define what a MAC is and how it is used. Two users who wish to communicate in an authenticated manner begin by generating and sharing a secret key k in advance of their communication. When one party wants to send a message m to the other, she computes a MAC tag (or simply a tag) t based on the message and the shared key, and sends the message m and the tag t to the other party. The tag is computed using a *tag-generation algorithm* denoted by Mac; thus, rephrasing what we have already said, the sender of a message m computes $t \leftarrow \mathsf{Mac}_k(m)$ and transmits (m, t) to the receiver. Upon receiving (m, t), the

[1] In the web-cookie example discussed earlier, the merchant is (in effect) communicating "with itself" with the user acting as a communication channel. In that setting, the server alone needs to know the key since it is acting as both sender and receiver.

second party *verifies* whether t is a valid tag on the message m (with respect to the shared key) or not. This is done by running a *verification algorithm* Vrfy that takes as input the shared key as well as a message m and a tag t, and indicates whether the given tag is valid. Formally:

DEFINITION 4.1 *A* message authentication code (*or* MAC) *consists of three probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *such that:*

1. *The* key-generation algorithm Gen *takes as input the security parameter* 1^n *and outputs a key* k *with* $|k| \geq n$.

2. *The* tag-generation algorithm Mac *takes as input a key* k *and a message* $m \in \{0,1\}^*$*, and outputs a tag* t*. Since this algorithm may be randomized, we write this as* $t \leftarrow \mathsf{Mac}_k(m)$.

3. *The deterministic* verification algorithm Vrfy *takes as input a key* k*, a message* m*, and a tag* t*. It outputs a bit* b*, with* $b = 1$ *meaning* valid *and* $b = 0$ *meaning* invalid*. We write this as* $b := \mathsf{Vrfy}_k(m, t)$.

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

If there is a function ℓ *such that for every* k *output by* $\mathsf{Gen}(1^n)$*, algorithm* Mac_k *is only defined for messages* $m \in \{0,1\}^{\ell(n)}$*, then we call the scheme a* fixed-length MAC for messages of length $\ell(n)$.

As with private-key encryption, $\mathsf{Gen}(1^n)$ almost always simply chooses a uniform key $k \in \{0,1\}^n$, and we omit Gen in that case.

Canonical verification. For deterministic message authentication codes (that is, where Mac is a deterministic algorithm), the canonical way to perform verification is to simply re-compute the tag and check for equality. In other words, $\mathsf{Vrfy}_k(m, t)$ first computes $\tilde{t} := \mathsf{Mac}_k(m)$ and then outputs 1 if and only if $\tilde{t} = t$. Even for deterministic MACs, however, it is useful to define a separate Vrfy algorithm in order to explicitly distinguish the semantics of *authenticating* a message vs. *verifying* its authenticity.

Security of Message Authentication Codes

We now define the default notion of security for message authentication codes. The intuitive idea behind the definition is that no efficient adversary should be able to generate a valid tag on any "new" message that was not previously sent (and authenticated) by one of the communicating parties.

As with any security definition, to formalize this notion we have to define both the adversary's power as well as what should be considered a "break." As usual, we consider only probabilistic polynomial-time adversaries[2] and

[2]See Section 4.6 for a discussion of information-theoretic message authentication, where no computational restrictions are placed on the adversary.

so the real question is how we model the adversary's interaction with the communicating parties. In the setting of message authentication, an adversary observing the communication between the honest parties may be able to see all the messages sent by these parties along with their corresponding MAC tags. The adversary may also be able to influence the *content* of these messages, whether directly or indirectly (if, e.g., external actions of the adversary affect the messages sent by the parties). This is true, for example, in the web cookie example from earlier, where the user's own actions influence the contents of the cookie being stored on his computer.

To formally model the above we allow the adversary to request MAC tags for *any* messages of its choice. Formally, we give the adversary access to a *MAC oracle* $\mathsf{Mac}_k(\cdot)$; the adversary can repeatedly submit any message m of its choice to this oracle, and is given in return a tag $t \leftarrow \mathsf{Mac}_k(m)$. (For a fixed-length MAC, only messages of the correct length can be submitted.)

We will consider it a "break" of the scheme if the adversary is able to output any message m along with a tag t such that: (1) t is a valid tag on the message m (i.e., $\mathsf{Vrfy}_k(m,t) = 1$), and (2) the adversary had not previously requested a MAC tag on the message m (i.e., from its oracle). The first condition means that if the adversary were to send (m,t) to one of the honest parties, then this party would be mistakenly fooled into thinking that m originated from the legitimate party since $\mathsf{Vrfy}_k(m,t) = 1$. The second condition is required because it is always possible for the adversary to just copy a message and MAC tag that were previously sent by one of the legitimate parties (and, of course, these would be accepted as valid). Such a *replay attack* is not considered a "break" of the message authentication code. This does *not* mean that replay attacks are not a security concern; they are, and we will have more to say about them below.

A MAC satisfying the level of security specified above is said to be *existentially unforgeable under an adaptive chosen-message attack*. "Existential unforgeability" refers to the fact that the adversary must not be able to forge a valid tag on *any* message, and "adaptive chosen-message attack" refers to the fact that the adversary is able to obtain MAC tags on arbitrary messages chosen adaptively during its attack.

Toward the formal definition, consider the following experiment for a message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$, an adversary $\mathcal{A}$, and value n for the security parameter:

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

1. *A key k is generated by running* $\mathsf{Gen}(1^n)$.
2. *The adversary $\mathcal{A}$ is given input* 1^n *and oracle access to* $\mathsf{Mac}_k(\cdot)$. *The adversary eventually outputs (m,t). Let $\mathcal{Q}$ denote the set of all queries that $\mathcal{A}$ asked its oracle.*
3. $\mathcal{A}$ *succeeds if and only if (1)* $\mathsf{Vrfy}_k(m,t) = 1$ *and (2)* $m \notin \mathcal{Q}$. *In that case the output of the experiment is defined to be* 1.

A MAC is secure if no efficient adversary can succeed in the above experiment with non-negligible probability:

DEFINITION 4.2 *A message authentication code* $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *is* existentially unforgeable under an adaptive chosen-message attack*, or just* secure*, if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that:*

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

Is the definition *too* strong? The above definition is rather strong in two respects. First, the adversary is allowed to request MAC tags for *any* messages of its choice. Second, the adversary is considered to have "broken" the scheme if it can output a valid tag on *any* previously unauthenticated message. One might object that both these components of the definition are unrealistic and overly strong: in "real-world" usage of a MAC, the honest parties would only authenticate "meaningful" messages (over which the adversary might have only limited control), and similarly it should only be considered a breach of security if the adversary can forge a valid tag on a "meaningful" message. Why not tailor the definition to capture this?

The crucial point is that what constitutes a meaningful message is entirely *application dependent*. While some applications of a MAC may only ever authenticate English-text messages, other applications may authenticate spreadsheet files, others database entries, and others raw data. Protocols may also be designed where *anything* will be authenticated—in fact, certain protocols for entity authentication do exactly this. By making the definition of security for MACs as strong as possible, we ensure that secure MACs are broadly applicable for a wide range of purposes, without having to worry about compatibility of the MAC with the semantics of the application.

Replay attacks. We emphasize that the above definition, and message authentication codes on their own, offer no protection against *replay attacks* whereby a previously sent message (and its MAC tag) are replayed to an honest party. Nevertheless, replay attacks are a serious concern! Consider again the scenario where a user (say, Alice) sends a request to her bank to transfer $1,000 from her account to some other user (say, Bob). In doing so, Alice can compute a MAC tag and append it to the request so the bank knows the request is authentic. If the MAC is secure, Bob will be unable to intercept the request and change the amount to $10,000 because this would involve forging a valid tag on a previously unauthenticated message. However, nothing prevents Bob from intercepting Alice's message and replaying it *ten times* to the bank. If the bank accepts each of these messages, the net effect is that $10,000 will be transferred to Bob's account rather than the desired $1,000.

Despite the real threat that replay attacks represent, a MAC by itself *cannot* protect against such attacks since the definition of a MAC (Definition 4.1)

does not incorporate any notion of *state* into the verification algorithm (and so every time a valid pair (m, t) is presented to the verification algorithm, it will always output 1). Rather, protection against replay attacks—if such protection is necessary at all—must be handled by some higher-level application. The reason the definition of a MAC is structured this way is, once again, because we are unwilling to assume any semantics regarding applications that use MACs; in particular, the decision as to whether or not a replayed message should be treated as "valid" may be application dependent.

Two common techniques for preventing replay attacks are to use *sequence numbers* (also known as *counters*) or *time-stamps*. The first approach, described (in a more general context) in Section 4.5.3, requires the communicating users to maintain (synchronized) state, and can be problematic when users communicate over a lossy channel where messages are occasionally dropped (though this problem can be mitigated). In the second approach, using time-stamps, the sender prepends the current time T (say, to the nearest millisecond) to the message before authenticating, and sends T along with the message and the resulting tag t. When the receiver obtains T, m, t, it verifies that t is a valid tag on $T \| m$ and that T is within some acceptable clock skew from the current time T' at the receiver. This method has certain drawbacks as well, including the need for the sender and receiver to maintain closely synchronized clocks, and the possibility that a replay attack can still take place if it is done quickly enough (specifically, within the acceptable time window).

Strong MACs. As defined, a secure MAC ensures that an adversary cannot generate a valid tag on a new message that was never previously authenticated. But it does not rule out the possibility that an attacker might be able to generate a new *tag* on a previously authenticated message. That is, a MAC guarantees that if an attacker learns tags $t_1, \ldots$ on messages $m_1, \ldots$, then it will not be able to forge a valid tag t on any message $m \notin \{m_1, \ldots\}$. However, it may be possible for an adversary to "forge" a *different* valid tag $t'_1 \neq t_1$ on the message m_1. In general, this type of adversarial behavior is not a concern. Nevertheless, in some settings it is useful to consider a stronger definition of security for MACs where such behavior is ruled out.

Formally, we consider a modified experiment $\mathsf{Mac\text{-}sforge}$ that is defined in exactly the same way as $\mathsf{Mac\text{-}forge}$, except that now the set $\mathcal{Q}$ contains *pairs* of oracle queries and their associated responses. (That is, $(m, t) \in \mathcal{Q}$ if $\mathcal{A}$ queried $\mathsf{Mac}_k(m)$ and received in response the tag t.) The adversary $\mathcal{A}$ succeeds (and experiment $\mathsf{Mac\text{-}sforge}$ evaluates to 1) if and only if $\mathcal{A}$ outputs (m, t) such that $\mathsf{Vrfy}_k(m, t) = 1$ and $(m, t) \notin \mathcal{Q}$.

DEFINITION 4.3 *A message authentication code* $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *is* strongly secure*, or a* strong MAC*, if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that:*

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

It is not hard to see that if a secure MAC uses canonical verification then it is also strongly secure. This is important since all real-world MACs use canonical verification. We leave the proof of the following as an exercise.

PROPOSITION 4.4 *Let* $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *be a secure MAC that uses canonical verification. Then* Π *is a strong MAC.*

Verification Queries

Definitions 4.2 and 4.3 give the adversary access to a MAC oracle, which corresponds to a real-world adversary who can influence an honest sender to generate a tag for some message m. One could also consider an adversary who interacts with an honest receiver, sending m', t' to the receiver to learn whether $\mathsf{Vrfy}_k(m', t') = 1$. Such an adversary could be captured formally in the natural way by giving the adversary in the above definitions access to a verification oracle as well.

A definition that incorporates a verification oracle in this way is, perhaps, the "right" way to define security for message authentication codes. It turns out, however, that for MACs that use canonical verification it makes no difference: any such MAC that satisfies Definition 4.2 also satisfies the definitional variant in which verification queries are allowed. Similarly, any strong MAC automatically also remains secure even in a setting where verification queries are possible. (This, in fact, serves as one motivation for the definition of strong security for MACs.) For MACs that do not use canonical verification, however, allowing verification queries can make a difference; see Exercises 4.2 and 4.3. Since most MACs covered in this book (as well as MACs used in practice) use canonical verification, we use the traditional definitions that omit access to a verification oracle.

A potential timing attack. One issue not addressed by the above is the possibility of carrying out a *timing attack* on MAC verification. Here, we consider an adversary who can send message/tag pairs to the receiver—thus using the receiver as a verification oracle—and learn not only whether the receiver accepts or rejects, but also the *time* it takes for the receiver to make this decision. We show that if such an attack is possible then a natural implementation of MAC verification leads to an easily exploitable vulnerability.

(Note that in our usual cryptographic definitions of security, the attacker learns only the output of the oracles it has access to, but nothing else. The attack we describe here, which is an example of a *side-channel attack*, shows that certain real-world attacks are not captured by the usual definitions.)

Concretely, assume a MAC using canonical verification. To verify a tag t on a message m, the receiver computes $t' := \mathsf{Mac}_k(m)$ and then compares t' to t, outputting 1 if and only if t' and t are equal. Assume this comparison is implemented using a standard routine (like `strcmp` in C) that compares t and t' one byte at a time, and rejects as soon as the first unequal byte is

encountered. The observation is that, when implemented in this way, the *time* to reject differs depending on the *position* of the first unequal byte.

It is possible to use this seemingly inconsequential information to forge a tag on any desired message m. The basic idea is this: say the attacker knows the first i bytes of the correct tag for m. (At the outset, $i = 0$.) The attacker will learn the next byte of the correct tag by sending $(m, t_0), \ldots, (m, t_{255})$ to the receiver, where t_j is the string with the first i bytes set correctly, the $(i+1)$st byte equal to j (in hexadecimal), and the remaining bytes set to `0x00`. All of these tags will likely be rejected (if not, then the attacker succeeds anyway); however, for exactly one of these tags the first $(i + 1)$ bytes will match the correct tag and rejection will take slightly longer than the rest. If t_j is the tag that caused rejection to take the longest, the attacker learns that the $(i+1)$st byte of the correct tag is j. In this way, the attacker learns each byte of the correct tag using at most 256 queries to the verification oracle. For a 16-byte tag, this attack requires only 4096 queries in the worst case.

One might wonder whether this attack is realistic, as it requires access to a verification oracle as well as the ability to measure the difference in time taken to compare i vs. $i + 1$ bytes. In fact, exactly such attacks have been carried out against real systems! As just one example, MACs were used to verify code updates in the Xbox 360, and the implementation of MAC verification used there had a difference of 2.2 milliseconds between rejection times. Attackers were able to exploit this and load pirated games onto the hardware.

Based on the above, we conclude that MAC verification should use *time-independent* string comparison that always compares *all* bytes.

4.3 Constructing Secure Message Authentication Codes

4.3.1 A Fixed-Length MAC

Pseudorandom functions are a natural tool for constructing secure message authentication codes. Intuitively, if the MAC tag t is obtained by applying a pseudorandom function to the message m, then forging a tag on a previously unauthenticated message requires the adversary to correctly guess the value of the pseudorandom function at a "new" input point. The probability of guessing the value of a *random* function on a new point is 2^{-n} (if the output length of the function is n). The probability of guessing such a value for a *pseudorandom* function can be only negligibly greater.

The above idea, shown in Construction 4.5, works for constructing a secure *fixed-length* MAC for messages of length n (since our pseudorandom functions by default have n-bit block length). This is useful, but falls short of our goal. In Section 4.3.2, we show how to extend this to handle messages of arbitrary length. We explore more efficient constructions of MACs for arbitrary-length messages in Sections 4.4 and 5.3.2.

CONSTRUCTION 4.5

Let F be a pseudorandom function. Define a fixed-length MAC for messages of length n as follows:

- Mac: on input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^n$, output the tag $t := F_k(m)$. (If $|m| \neq |k|$ then output nothing.)
- Vrfy: on input a key $k \in \{0,1\}^n$, a message $m \in \{0,1\}^n$, and a tag $t \in \{0,1\}^n$, output 1 if and only if $t \stackrel{?}{=} F_k(m)$. (If $|m| \neq |k|$, then output 0.)

A fixed-length MAC from any pseudorandom function.

THEOREM 4.6 *If F is a pseudorandom function, then Construction 4.5 is a secure fixed-length MAC for messages of length n.*

PROOF 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. Consider the message authentication code $\widetilde{\Pi} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Mac}}, \widetilde{\mathsf{Vrfy}})$ which is the same as $\Pi = (\mathsf{Mac}, \mathsf{Vrfy})$ in Construction 4.5 except that a truly random function f is used instead of the pseudorandom function F_k. That is, $\widetilde{\mathsf{Gen}}(1^n)$ works by choosing a uniform function $f \in \mathsf{Func}_n$, and $\widetilde{\mathsf{Mac}}$ computes a tag just as Mac does except that f is used instead of F_k. It is immediate that

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1] \leq 2^{-n} \tag{4.1}$$

because for any message $m \notin \mathcal{Q}$, the value $t = f(m)$ is uniformly distributed in $\{0,1\}^n$ from the point of view of the adversary $\mathcal{A}$.

We next show that there is a negligible function negl such that

$$\left|\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] - \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1]\right| \leq \mathsf{negl}(n); \tag{4.2}$$

combined with Equation (4.1), this shows that

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

proving the theorem.

To prove Equation (4.2), we construct a polynomial-time distinguisher D that is given oracle access to some function, and whose goal is to determine whether this function is pseudorandom (i.e., equal to F_k for uniform $k \in \{0,1\}^n$) or random (i.e., equal to f for uniform $f \in \mathsf{Func}_n$). To do this, D emulates the message authentication experiment for $\mathcal{A}$ and observes whether $\mathcal{A}$ succeeds in outputting a valid tag on a "new" message. If so, D guesses that its oracle is a pseudorandom function; otherwise, D guesses that its oracle is a random function. In detail:

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

1. Run $\mathcal{A}(1^n)$. Whenever $\mathcal{A}$ queries its MAC oracle on a message m (i.e., whenever $\mathcal{A}$ requests a tag on a message m), answer this query in the following way:

   Query $\mathcal{O}$ with m and obtain response t; return t to $\mathcal{A}$.
2. When $\mathcal{A}$ outputs (m, t) at the end of its execution, do:
   (a) Query $\mathcal{O}$ with m and obtain response $\hat{t}$.
   (b) If (1) $\hat{t} = t$ and (2) $\mathcal{A}$ never queried its MAC oracle on m, then output 1; otherwise, output 0.

It is clear that D runs in polynomial time.

Notice that if D's oracle is a pseudorandom function, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$. Furthermore, D outputs 1 exactly when $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$. Therefore

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

where $k \in \{0,1\}^n$ is chosen uniformly in the above. If D's oracle is a random function, then the view of $\mathcal{A}$ when run as a sub-routine by D is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n)$, and again D outputs 1 exactly when $\mathsf{Mac\text{-}forge}_{\mathcal{A},\widetilde{\Pi}}(n) = 1$. Thus,

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

where $f \in \mathsf{Func}_n$ is chosen uniformly.

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

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

This implies Equation (4.2), completing the proof of the theorem. ■

4.3.2 Domain Extension for MACs

Construction 4.5 is important in that it shows a general paradigm for constructing secure message authentication codes from pseudorandom functions. Unfortunately, the construction is only capable of handling *fixed-length* messages that are furthermore rather short.[3] These limitations are unacceptable

[3]Given a pseudorandom function taking arbitrary-length inputs, Construction 4.5 would yield a secure MAC for messages of arbitrary length. Likewise, a pseudorandom function

in most applications. We show here how a general MAC, handling arbitrary-length messages, can be constructed from any fixed-length MAC for messages of length n. The construction we show is not very efficient and is unlikely to be used in practice. Indeed, far more efficient constructions of secure MACs are known, as we discuss in Sections 4.4 and 5.3.2. We include the present construction for its simplicity and generality.

Let $\Pi' = (\mathsf{Mac}', \mathsf{Vrfy}')$ be a secure fixed-length MAC for messages of length n. Before presenting the construction of a MAC for arbitrary-length messages based on Π', we rule out some simple ideas and describe some canonical attacks that must be prevented. Below, we parse the message m to be authenticated as a sequence of blocks $m_1, \ldots, m_d$; note that, since our aim is to handle messages of arbitrary length, d can vary from message to message.

1. A natural first idea is to simply authenticate each block separately, i.e., compute $t_i := \mathsf{Mac}'_k(m_i)$ for all i, and output $\langle t_1, \ldots, t_d \rangle$ as the tag. This prevents an adversary from sending any previously unauthenticated block without being detected. However, it does not prevent a *block reordering attack* in which the attacker shuffles the order of blocks in an authenticated message. Specifically, if $\langle t_1, t_2 \rangle$ is a valid tag on the message m_1, m_2 (with $m_1 \neq m_2$), then $\langle t_2, t_1 \rangle$ is a valid tag on the (different) message m_2, m_1 (something that is not allowed by Definition 4.2).

2. We can prevent the previous attack by authenticating a block index along with each block. That is, we now compute $t_i = \mathsf{Mac}'_k(i \| m_i)$ for all i, and output $\langle t_1, \ldots, t_d \rangle$ as the tag. (Note that the block length $|m_i|$ will have to change.) This does not prevent a *truncation attack* whereby an attacker simply drops blocks from the end of the message (and drops the corresponding blocks of the tag as well).

3. The truncation attack can be thwarted by additionally authenticating the message length along with each block. (Authenticating the message length as a separate block does not work. Do you see why?) That is, compute $t_i = \mathsf{Mac}'_k(\ell \| i \| m_i)$ for all i, where ℓ denotes the length of the message in bits. (Once again, the block length $|m_i|$ will need to decrease.) This scheme is vulnerable to a *"mix-and-match" attack* where the adversary combines blocks from different messages. For example, if the adversary obtains tags $\langle t_1, \ldots, t_d \rangle$ and $\langle t'_1, \ldots, t'_d \rangle$ on messages $m = m_1, \ldots, m_d$ and $m' = m'_1, \ldots, m'_d$, respectively, it can output the valid tag $\langle t_1, t'_2, t_3, t'_4, \ldots \rangle$ on the message $m_1, m'_2, m_3, m'_4, \ldots$.

We can prevent this last attack by also including a random "message identifier" along with each block that prevents blocks from different messages from being combined. This leads us to Construction 4.7.

---

with a larger domain would yield a secure MAC for longer messages. However, existing *practical* pseudorandom functions (i.e., block ciphers) take short, fixed-length inputs.

CONSTRUCTION 4.7

Let $\Pi' = (\mathsf{Mac}', \mathsf{Vrfy}')$ be a fixed-length MAC for messages of length n. Define a MAC as follows:

- Mac: on input a key $k \in \{0,1\}^n$ and a message $m \in \{0,1\}^*$ of (nonzero) length $\ell < 2^{n/4}$, parse m into d blocks $m_1, \ldots, m_d$, each of length $n/4$. (The final block is padded with 0s if necessary.) Choose a uniform identifier $r \in \{0,1\}^{n/4}$.

  For $i = 1, \ldots, d$, compute $t_i \leftarrow \mathsf{Mac}'_k(r \| \ell \| i \| m_i)$, where i, ℓ are encoded as strings of length $n/4$.[†] Output the tag $t := \langle r, t_1, \ldots, t_d \rangle$.
- Vrfy: on input a key $k \in \{0,1\}^n$, a message $m \in \{0,1\}^*$ of length $\ell < 2^{n/4}$, and a tag $t = \langle r, t_1, \ldots, t_{d'} \rangle$, parse m into d blocks $m_1, \ldots, m_d$, each of length $n/4$. (The final block is padded with 0s if necessary.) Output 1 if and only if $d' = d$ and $\mathsf{Vrfy}'_k(r \| \ell \| i \| m_i, t_i) = 1$ for $1 \le i \le d$.

[†] Note that i and ℓ can be encoded using $n/4$ bits because $i, \ell < 2^{n/4}$.

A MAC for arbitrary-length messages from any fixed-length MAC.

(Technically, the scheme only handles messages of length less than $2^{n/4}$. Asymptotically, since this is an exponential bound, honest parties will not authenticate messages that long and any polynomial-time adversary could not submit messages that long to its MAC oracle. In practice, when a concrete value of n is fixed, one must ensure that this bound is acceptable.)

THEOREM 4.8 *If Π' is a secure fixed-length MAC for messages of length n, then Construction 4.7 is a secure MAC (for arbitrary-length messages).*

PROOF The intuition is that as long as Π' is secure, an adversary cannot introduce a new block with a valid tag. Furthermore, the extra information included in each block prevents the various attacks (dropping blocks, re-ordering blocks, etc.) sketched earlier. We will prove security by essentially showing that these attacks are the only ones possible.

Let Π be the MAC given by Construction 4.7, and let $\mathcal{A}$ be a probabilistic polynomial-time adversary. We show that $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1]$ is negligible. We first introduce some notation that will be used in the proof. Let Repeat denote the event that the same random identifier appears in two of the tags returned by the MAC oracle in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$. Letting $(m, t = \langle r, t_1, \ldots \rangle)$ denote the final output of $\mathcal{A}$, where $m = m_1, \ldots$ has length ℓ, we let $\mathsf{NewBlock}$ be the event that at least one of the blocks $r \| \ell \| i \| m_i$ was never previously authenticated by Mac' in the course of answering $\mathcal{A}$'s Mac queries. (Note that, by construction of Π, it is easy to tell exactly which blocks are authenticated by Mac'_k when computing $\mathsf{Mac}_k(m)$.) Informally, $\mathsf{NewBlock}$ is the event that $\mathcal{A}$ tries to output a valid tag on a block that was

never authenticated by the underlying fixed-length MAC Π'.

We have

$$\begin{aligned}\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1] &= \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \mathsf{Repeat}] \\ &\quad + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \mathsf{NewBlock}] \\ &\quad + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \overline{\mathsf{NewBlock}}] \\ &\leq \Pr[\mathsf{Repeat}] \qquad (4.3) \\ &\quad + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \mathsf{NewBlock}] \\ &\quad + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \overline{\mathsf{NewBlock}}].\end{aligned}$$

We show that the first two terms of Equation (4.3) are negligible, and the final term is 0. This implies $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1]$ is negligible, as desired.

CLAIM 4.9 $\Pr[\mathsf{Repeat}]$ *is negligible.*

PROOF Let $q(n)$ be the number of MAC oracle queries made by $\mathcal{A}$. To answer the ith oracle query of $\mathcal{A}$, the oracle chooses r_i uniformly from a set of size $2^{n/4}$. The probability of event Repeat is exactly the probability that $r_i = r_j$ for some $i \neq j$. Applying the "birthday bound" (Lemma A.15), we have that $\Pr[\mathsf{Repeat}] \leq \frac{q(n)^2}{2^{n/4}}$. Since $\mathcal{A}$ makes only polynomially many queries, this value is negligible. ∎

We next consider the final term on the right-hand side of Equation (4.3). We argue that if $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1$, but Repeat did not occur, then it must be the case that NewBlock occurred. That is, $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}}$ implies NewBlock, and so

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)=1 \wedge \overline{\mathsf{Repeat}} \wedge \overline{\mathsf{NewBlock}}] = 0.$$

This is, in some sense, the heart of the proof.

Again let $q = q(n)$ denote the number of MAC oracle queries made by $\mathcal{A}$, and let r_i denote the random identifier used to answer the ith oracle query of $\mathcal{A}$. If Repeat does not occur then the values $r_1, \ldots, r_q$ are distinct. Let $(m, t = \langle r, t_1, \ldots \rangle)$ be the output of $\mathcal{A}$, with $m = m_1, \ldots$. If $r \notin \{r_1, \ldots, r_q\}$, then NewBlock clearly occurs. If not, then $r = r_j$ for some unique j, and the blocks $r \| \ell \| 1 \| m_1, \ldots$ could then not possibly have been authenticated during the course of answering any Mac queries other than the jth such query. Let $m^{(j)}$ be the message that was used by $\mathcal{A}$ for its jth oracle query, and let ℓ_j be its length. There are two cases to consider:

Case 1: $\ell \neq \ell_j$**.** The blocks authenticated when answering the jth Mac query all have $\ell_j \neq \ell$ in the second position. So $r \| \ell \| 1 \| m_1$, in particular, was never authenticated in the course of answering the jth Mac query, and NewBlock occurs.

Case 2: $\ell = \ell_j$**.** If $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$, then we must have $m \neq m^{(j)}$. Let $m^{(j)} = m_1^{(j)}, \ldots$. Since m and $m^{(j)}$ have equal length, there must be at least one index i for which $m_i \neq m_i^{(j)}$. The block $r\|\ell\|i\|m_i$ was then never authenticated in the course of answering the jth Mac query. (Because i is included in the third position of the block, the block $r\|\ell\|i\|m_i$ could only possibly have been authenticated if $r\|\ell\|i\|m_i = r_j\|\ell_j\|i\|m_i^{(j)}$, but this is not true since $m_i \neq m_i^{(j)}$.)

To complete the proof of the theorem, we bound the second term on the right-hand side of Equation (4.3):

CLAIM 4.10 $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \mathsf{NewBlock}]$ *is negligible.*

The claim relies on security of Π'. We construct a PPT adversary $\mathcal{A}'$ who attacks the fixed-length MAC Π' and succeeds in outputting a valid forgery on a previously unauthenticated message with probability

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1] \geq \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1 \wedge \mathsf{NewBlock}]. \quad (4.4)$$

Security of Π' means that the left-hand side is negligible, proving the claim.

The construction of $\mathcal{A}'$ is the obvious one and so we describe it briefly. $\mathcal{A}'$ runs $\mathcal{A}$ as a sub-routine, and answers the request by $\mathcal{A}$ for a tag on m by choosing $r \leftarrow \{0,1\}^{n/4}$ itself, parsing m appropriately, and making the necessary queries to its own MAC oracle $\mathsf{Mac}'_k(\cdot)$. When $\mathcal{A}$ outputs $(m, t = \langle r, t_1, \ldots\rangle)$, then $\mathcal{A}'$ checks whether $\mathsf{NewBlock}$ occurs (this is easy to do since $\mathcal{A}'$ can keep track of all the queries it makes to its own oracle). If so, then $\mathcal{A}'$ finds the first block $r\|\ell\|i\|m_i$ that was never previously authenticated by Mac' and outputs $(r\|\ell\|i\|m_i, t_i)$. (If not, $\mathcal{A}'$ outputs nothing.)

The view of $\mathcal{A}$ when run as a sub-routine by $\mathcal{A}'$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$, and so the probabilities of events $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$ and $\mathsf{NewBlock}$ do not change. If $\mathsf{NewBlock}$ occurs then $\mathcal{A}'$ outputs a block $r\|\ell\|i\|m_i$ that was never previously authenticated by its own MAC oracle; if $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$ then the tag on every block is valid (with respect to Π'), and so in particular this is true for the block output by $\mathcal{A}'$. This means that whenever $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1$ and $\mathsf{NewBlock}$ occur we have $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1$, proving Equation (4.4). ■

4.4 CBC-MAC

Theorems 4.6 and 4.8 show that it is possible to construct a secure message authentication code for arbitrary-length messages from a pseudorandom

function taking inputs of fixed length n. This demonstrates, in principle, that secure MACs can be constructed from block ciphers. Unfortunately, the resulting construction is extremely inefficient: to compute a tag on a message of length dn, the block cipher is evaluated $4d$ times; the tag is more than $4dn$ bits long. Fortunately, far more efficient constructions are available. We explore one such construction here that relies solely on block ciphers, and another in Section 5.3.2 that uses an additional cryptographic primitive.

4.4.1 The Basic Construction

CBC-MAC is a standardized message authentication code used widely in practice. A basic version of CBC-MAC, secure when authenticating messages of any *fixed* length, is given as Construction 4.11. (See also Figure 4.1.) We caution that this basic scheme is *not* secure in the general case when messages of different lengths may be authenticated; see further discussion below.

CONSTRUCTION 4.11

Let F be a pseudorandom function, and fix a length function $\ell > 0$. The basic CBC-MAC construction is as follows:

- Mac: on input a key $k \in \{0,1\}^n$ and a message m of length $\ell(n) \cdot n$, do the following (we set $\ell = \ell(n)$ in what follows):

  1. Parse m as $m = m_1, \ldots, m_\ell$ where each m_i is of length n.
  2. Set $t_0 := 0^n$. Then, for $i = 1$ to ℓ:
     Set $t_i := F_k(t_{i-1} \oplus m_i)$.

  Output t_ℓ as the tag.

- Vrfy: on input a key $k \in \{0,1\}^n$, a message m, and a tag t, do: If m is not of length $\ell(n) \cdot n$ then output 0. Otherwise, output 1 if and only if $t \stackrel{?}{=} \mathsf{Mac}_k(m)$.

Basic CBC-MAC (for fixed-length messages).

THEOREM 4.12 *Let ℓ be a polynomial. If F is a pseudorandom function, then Construction 4.11 is a secure MAC for messages of length $\ell(n) \cdot n$.*

The proof of Theorem 4.12 is somewhat involved. In the following section we will prove a more general result from which the above theorem follows.

Although Construction 4.11 can be extended in the obvious way to handle messages whose length is an arbitrary multiple of n, the construction is only secure when the length of the messages being authenticated is fixed and agreed upon in advance by the sender and receiver. (See Exercise 4.13.)

The advantage of this construction over Construction 4.5, which also gives a fixed-length MAC, is that the present construction can authenticate longer messages. Compared to Construction 4.7, CBC-MAC is much more efficient, requiring only d block-cipher evaluations for a message of length dn, and with a tag of length n only.

CBC-MAC vs. CBC-mode encryption. CBC-MAC is similar to the CBC mode of operation. There are, however, some important differences:

1. CBC-mode encryption uses a *random IV* and this is crucial for security. In contrast, CBC-MAC uses no IV (alternately, it can be viewed as using the fixed value $IV = 0^n$) and this is also crucial for security. Specifically, CBC-MAC using a random IV is not secure.

2. In CBC-mode encryption all intermediate values t_i (called c_i in the case of CBC-mode encryption) are output by the encryption algorithm as part of the ciphertext, whereas in CBC-MAC only the final block is output as the tag. If CBC-MAC is modified to output all the $\{t_i\}$ obtained during the course of the computation then it is no longer secure.

In Exercise 4.14 you are asked to verify that the modifications of CBC-MAC discussed above are insecure. These examples illustrate the fact that harmless-looking modifications to cryptographic constructions can render them insecure. One should always implement a cryptographic construction exactly as specified and not introduce any variations (unless the variations themselves can be proven secure). Furthermore, it is essential to understand the construction being used. In many cases a cryptographic library provides a programmer with a "CBC function," but does not distinguish between the use of this function for encryption or message authentication.

Secure CBC-MAC for arbitrary-length messages. We briefly describe two ways Construction 4.11 can be modified, in a provably secure fashion, to handle arbitrary-length messages. (Here for simplicity we assume that all messages being authenticated have length a multiple of n, and that Vrfy rejects

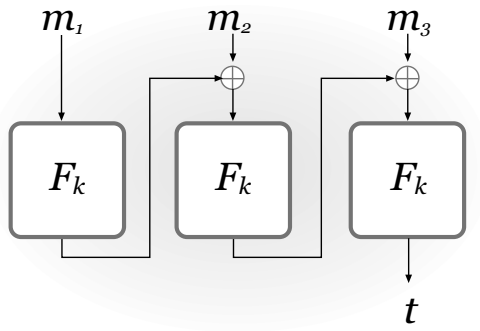


FIGURE 4.1: Basic CBC-MAC (for fixed-length messages).

any message whose length is not a multiple of n. In the following section we treat the more general case where messages can have arbitrary length.)

1. *Prepend* the message m with its length $|m|$ (encoded as an n-bit string), and then compute basic CBC-MAC on the result; see Figure 4.2. Security of this variant follows from the results proved in the next section.

   Note that appending $|m|$ to the *end* of the message and then computing the basic CBC-MAC is *not* secure.

2. Change the scheme so that key generation chooses two independent, uniform keys $k_1 \in \{0,1\}^n$ and $k_2 \in \{0,1\}^n$. Then to authenticate a message m, first compute the basic CBC-MAC of m using k_1 and let t be the result; output the tag $\hat{t} := F_{k_2}(t)$.

The second option has the advantage of not needing to know the message length in advance (i.e., when beginning to compute the tag). However, it has the drawback of using two keys for F. Note that, 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. Despite this, in practice, the operation of initializing a key for a block cipher is considered relatively expensive. Therefore, requiring two different keys—even if they are derived on the fly—is less desirable.

4.4.2 *Proof of Security

In this section we prove security of different variants of CBC-MAC. We begin by summarizing the results, and then give the details of the proof. Before beginning, we remark that the proof in this section is quite involved, and is intended for advanced readers.

Throughout this section, fix a keyed function F that, for security parameter n, maps n-bit keys and n-bit inputs to n-bit outputs. We define a keyed

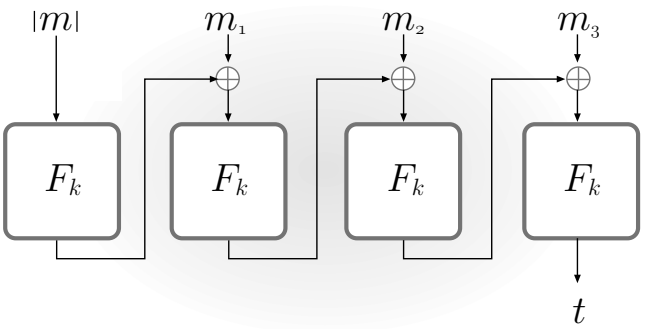


FIGURE 4.2: A variant of CBC-MAC secure for authenticating arbitrary-length messages.

function CBC that, for security parameter n, maps n-bit keys and inputs in $(\{0,1\}^n)^*$ (i.e., strings whose length is a multiple of n) to n-bit outputs. This function is defined as

$$\mathsf{CBC}_k(x_1,\ldots,x_\ell) \stackrel{\text{def}}{=} F_k\left(F_k\left(\cdots F_k(F_k(x_1)\oplus x_2)\oplus\cdots\right)\oplus x_\ell\right),$$

where $|x_1| = \cdots = |x_\ell| = |k| = n$. (We leave CBC_k undefined on the empty string.) Note that CBC is exactly basic CBC-MAC, although here we consider inputs of different lengths.

A set of strings $P \subset (\{0,1\}^n)^*$ is *prefix-free* if it does not contain the empty string, and no string $X \in P$ is a prefix of any other string $X' \in P$. We show:

THEOREM 4.13 *If F is a pseudorandom function, then CBC is a pseudorandom function as long as the set of inputs on which it is queried is prefix-free. Formally, for all probabilistic polynomial-time distinguishers D that query their oracle on a prefix-free set of inputs, there is a negligible function negl such that*

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

where k is chosen uniformly from $\{0,1\}^n$ and f is chosen uniformly from the set of functions mapping $(\{0,1\}^n)^*$ to $\{0,1\}^n$ (i.e., the value of f at each input is uniform and independent of the values of f at all other inputs).

Thus, we can convert a pseudorandom function F for fixed-length inputs into a pseudorandom function CBC for arbitrary-length inputs (subject to a constraint on which inputs can be queried)! To use this for message authentication, we adapt the idea of Construction 4.5 as follows: to authenticate a message m, first apply some *encoding function* encode to obtain a (nonempty) string $\mathsf{encode}(m) \in (\{0,1\}^n)^*$; then output the tag $\mathsf{CBC}_k(\mathsf{encode}(m))$. For this to be secure (cf. the proof of Theorem 4.6), the encoding needs to be *prefix-free*, namely, to have the property that for any distinct (legal) messages m_1, m_2, the string $\mathsf{encode}(m_1)$ is not a prefix of $\mathsf{encode}(m_2)$. This implies that for any set of (legal) messages $\{m_1,\ldots\}$, the set of encoded messages $\{\mathsf{encode}(m_1),\ldots\}$ is prefix-free.

We now examine two concrete applications of this idea:

- Fix ℓ, and let the set of legal messages be $\{0,1\}^{\ell(n)\cdot n}$. Then we can take the trivial encoding $\mathsf{encode}(m) = m$, which is prefix-free since one string cannot be a prefix of a different string of the same length. This is exactly basic CBC-MAC, and what we have said above implies that basic CBC-MAC is secure for messages of any fixed length (cf. Theorem 4.12).
- One way of handling arbitrary-length (nonempty) messages (technically, messages of length less than 2^n) is to encode a string $m \in \{0,1\}^*$ by prepending its length $|m|$ (encoded as an n-bit string), and then appending as many 0s as needed to make the length of the resulting

string a multiple of n. (This is essentially what is shown in Figure 4.2.) This encoding is prefix-free, and we therefore obtain a secure MAC for arbitrary-length messages.

The rest of this section is devoted to a proof of Theorem 4.13. In proving the theorem, we analyze CBC when it is "keyed" with a *random function* g rather than a random key k for some underlying pseudorandom function F. That is, we consider the keyed function CBC_g defined as

$$\mathsf{CBC}_g(x_1, \ldots, x_\ell) \stackrel{\text{def}}{=} g\left(g\left(\cdots g(g(x_1) \oplus x_2) \oplus \cdots\right) \oplus x_\ell\right)$$

where, for security parameter n, the function g maps n-bit inputs to n-bit outputs, and $|x_1| = \cdots = |x_\ell| = n$. Note that CBC_g as defined here is not efficient (since the representation of g requires space exponential in n); nevertheless, it is still a well-defined, keyed function.

We show that if g is chosen uniformly from Func_n, then CBC_g is indistinguishable from a random function mapping $(\{0,1\}^n)^*$ to n-bit strings, as long as a prefix-free set of inputs is queried. More precisely:

CLAIM 4.14 *Fix any $n \geq 1$. For all distinguishers D that query their oracle on a prefix-free set of q inputs, where the longest such input contains ℓ blocks, it holds that:*

$$\left| \Pr[D^{\mathsf{CBC}_g(\cdot)}(1^n) = 1] - \Pr[D^{f(\cdot)}(1^n) = 1] \right| \leq \frac{q^2\ell^2}{2^n},$$

where g is chosen uniformly from Func_n, and f is chosen uniformly from the set of functions mapping $(\{0,1\}^n)^*$ to $\{0,1\}^n$.

(The claim is unconditional, and does not impose any constraints on the running time of D. Thus we may take D to be deterministic.) The above implies Theorem 4.13 using standard techniques that we have already seen. In particular, for any D running in polynomial time we must have $q(n), \ell(n) = \mathsf{poly}(n)$ and so $q(n)^2\ell(n)^2 \cdot 2^{-n}$ is negligible.

PROOF (of Claim 4.14) Fix some $n \geq 1$. The proof proceeds in two steps: We first define a notion of *smoothness* and prove that CBC is smooth; we then show that smoothness implies the claim.

Let $P = \{X_1, \ldots, X_q\}$ be a prefix-free set of q inputs, where each X_i is in $(\{0,1\}^n)^*$ and the longest string in P contains ℓ blocks (i.e., each $X_i \in P$ contains at most ℓ blocks of length n). Note that for any $t_1, \ldots, t_q \in \{0,1\}^n$ it holds that $\Pr[\forall i : f(X_i) = t_i] = 2^{-nq}$, where the probability is over uniform choice of the function f from the set of functions mapping $(\{0,1\}^n)^*$ to $\{0,1\}^n$. We say that CBC is (q, ℓ, δ)-*smooth* if for every prefix-free set $P = \{X_1, \ldots, X_q\}$ as above and every $t_1, \ldots, t_q \in \{0,1\}^n$, it holds that

$$\Pr[\forall i : \mathsf{CBC}_g(X_i) = t_i] \geq (1 - \delta) \cdot 2^{-nq},$$

where the probability is over uniform choice of $g \in \mathsf{Func}_n$.

In words, CBC is smooth if for every fixed set of input/output pairs $\{(X_i, t_i)\}$, where the $\{X_i\}$ form a prefix-free set, the probability that $\mathsf{CBC}_g(X_i) = t_i$ for all i is δ-close to the probability that $f(X_i) = t_i$ for all i (where g is a random function from $\{0,1\}^n$ to $\{0,1\}^n$, and f is a random function from $(\{0,1\}^n)^*$ to $\{0,1\}^n$).

CLAIM 4.15 *CBC_g is (q, ℓ, δ)-smooth, for $\delta = q^2\ell^2 \cdot 2^{-n}$.*

PROOF For any $X \in (\{0,1\}^n)^*$, with $X = x_1, \ldots$ and $x_i \in \{0,1\}^n$, let $\mathcal{C}_g(X)$ denote the set of inputs on which g is evaluated during the computation of $\mathsf{CBC}_g(X)$; i.e., if $X \in (\{0,1\}^n)^m$ then

$$\mathcal{C}_g(X) \stackrel{\text{def}}{=} (x_1,\ \mathsf{CBC}_g(x_1) \oplus x_2,\ \ldots,\ \mathsf{CBC}_g(x_1, \ldots, x_{m-1}) \oplus x_m)\,.$$

For $X \in (\{0,1\}^n)^m$ and $X' \in (\{0,1\}^n)^{m'}$, with $\mathcal{C}_g(X) = (I_1, \ldots, I_m)$ and $\mathcal{C}_g(X') = (I'_1, \ldots, I'_{m'})$, say there is a *non-trivial collision in X* if $I_i = I_j$ for some $i \neq j$, and say that there is a *non-trivial collision between X and X'* if $I_i = I'_j$ but $(x_1, \ldots, x_i) \neq (x'_1, \ldots, x'_j)$ (in this latter case i may equal j). We say that there is a non-trivial collision in P if there is a non-trivial collision in some $X \in P$ or between some pair of strings $X, X' \in P$. Let Coll be the event that there is a non-trivial collision in P.

We prove the claim in two steps. First, we show that conditioned on there not being a non-trivial collision in P, the probability that $\mathsf{CBC}_g(X_i) = t_i$ for all i is exactly 2^{-nq}. Next, we show that the probability that there is a non-trivial collision in P is less than $\delta = q^2\ell^2 \cdot 2^{-n}$.

Consider choosing a uniform g by choosing, one-by-one, uniform values for the outputs of g on different inputs. Determining whether there is a non-trivial collision between two strings $X, X' \in P$ can be done by first choosing the values of $g(I_1)$ and $g(I'_1)$ (if $I'_1 = I_1$, these values are the same), then choosing values for $g(I_2)$ and $g(I'_2)$ (note that $I_2 = g(I_1) \oplus x_2$ and $I'_2 = g(I'_1) \oplus x'_2$ are defined once $g(I_1), g(I'_1)$ have been fixed), and continuing in this way until we choose values for $g(I_{m-1})$ and $g(I'_{m'-1})$. In particular, the values of $g(I_m), g(I'_{m'})$ need not be chosen in order to determine whether there is a non-trivial collision between X and X'. Continuing this line of reasoning, it is possible to determine whether Coll occurs by choosing the values of g on all but the final entries of each of $\mathcal{C}_g(X_1), \ldots, \mathcal{C}_g(X_q)$.

Assume Coll has not occurred after fixing the values of g on various inputs as described above. Consider the final entries in each of $\mathcal{C}_g(X_1), \ldots, \mathcal{C}_g(X_q)$. These entries are all distinct (this is immediate from the fact that Coll has not occurred), and we claim that the value of g on each of those points has not yet been fixed. Indeed, the only way the value of g could already be fixed on any of those points is if the final entry I_m of some $\mathcal{C}_g(X)$ is equal to a non-final entry I_j of some $\mathcal{C}_g(X')$. But since Coll has not occurred, this can

only happen if $X \neq X'$ and $(x'_1, \ldots, x'_j) = (x_1, \ldots, x_m)$. But then X would be a prefix of X', violating the assumption that P is prefix-free.

Since g is a random function, the above means that $\mathsf{CBC}_g(X_1), \ldots, \mathsf{CBC}_g(X_q)$ are *uniform* and *independent* of each other as well as all the other values of g that have already been fixed. (This is because $\mathsf{CBC}_g(X_i)$ is the value of g when evaluated at the final entry of $\mathcal{C}_g(X_i)$, an input value which is different from all the other inputs at which g has already been fixed.) Thus, for any $t_1, \ldots, t_q \in \{0,1\}^n$ we have:

$$\Pr\left[\forall i : \mathsf{CBC}_g(X_i) = t_i \mid \overline{\mathsf{Coll}}\right] = 2^{-nq}. \tag{4.5}$$

We next show that $\overline{\mathsf{Coll}}$ occurs with high probability by upper-bounding $\Pr[\mathsf{Coll}]$. For distinct $X_i, X_j \in P$, let $\mathsf{Coll}_{i,j}$ denote the event that there is a non-trivial collision in X or X', or a non-trivial collision between X and X'. We have $\mathsf{Coll} = \bigvee_{i,j} \mathsf{Coll}_{i,j}$ and so a union bound gives

$$\Pr[\mathsf{Coll}] \leq \sum_{i,j:\ i<j} \Pr[\mathsf{Coll}_{i,j}] = \binom{q}{2} \cdot \Pr[\mathsf{Coll}_{i,j}] \leq \frac{q^2}{2} \cdot \Pr[\mathsf{Coll}_{i,j}]. \tag{4.6}$$

Fixing distinct $X = X_i$ and $X' = X_j$ in P, we now bound $\mathsf{Coll}_{i,j}$. As will be clear from the analysis, the probability is maximized when X and X' are both as long as possible, and thus we assume they are each ℓ blocks long. Let $X = (x_1, \ldots, x_\ell)$ and $X' = (x'_1, \ldots, x'_\ell)$, and let t be the largest integer such that $(x_1, \ldots, x_t) = (x'_1, \ldots, x'_t)$. (Note that $t < \ell$ or else $X = X'$.) We assume $t > 0$, but the analysis below can be easily modified, giving the same result, if $t = 0$. We continue to let $I_1, I_2, \ldots$ (resp., $I'_1, I'_2, \ldots$) denote the inputs to g during the course of computing $\mathsf{CBC}_g(X)$ (resp., $\mathsf{CBC}_g(X')$); note that $(I'_1, \ldots, I'_t) = (I_1, \ldots, I_t)$. Consider choosing g by choosing uniform values for the outputs of g, one-by-one. We do this in $2\ell - 2$ steps as follows:

Steps 1 through $t-1$ (if $t > 1$): In each step i, choose a uniform value for $g(I_i)$, thus defining I_{i+1} and I'_{i+1} (which are equal).

Step t: Choose a uniform value for $g(I_t)$, thus defining I_{t+1} and I'_{t+1}.

Steps $t+1$ to $\ell-1$ (if $t < \ell - 1$): Choose, in turn, uniform values for each of $g(I_{t+1})$, $g(I_{t+2})$, $\ldots$, $g(I_{\ell-1})$, thus defining I_{t+2}, I_{t+3}, $\ldots$, I_ℓ.

Steps ℓ to $2\ell-2$ (if $t < \ell - 1$): Choose, in turn, uniform values for each of $g(I'_{t+1})$, $g(I'_{t+2})$, $\ldots$, $g(I'_{\ell-1})$, thus defining I'_{t+2}, I'_{t+3}, $\ldots$, I'_ℓ.

Let $\mathsf{Coll}(k)$ be the event that a non-trivial collision occurs by step k. Then

$$\Pr[\mathsf{Coll}_{i,j}] = \Pr\left[\bigvee_k \mathsf{Coll}(k)\right] \leq \Pr[\mathsf{Coll}(1)] + \sum_{k=2}^{2\ell-2} \Pr[\mathsf{Coll}(k) \mid \overline{\mathsf{Coll}(k-1)}], \tag{4.7}$$

using Proposition A.9. For $k < t$, we claim $\Pr[\mathsf{Coll}(k) \mid \overline{\mathsf{Coll}(k-1)}] = k/2^n$: indeed, if no non-trivial collision has occurred by step $k-1$, the value of

$g(I_k)$ is chosen uniformly in step k; a non-trivial collision occurs only if it happens that $I_{k+1} = g(I_k) \oplus x_{k+1}$ is equal to one of $\{I_1, \ldots, I_k\}$ (which are all distinct, since $\overline{\mathsf{Coll}(k-1)}$ has not occurred). By similar reasoning, we have $\Pr[\mathsf{Coll}(t) \mid \overline{\mathsf{Coll}(t-1)}] \leq 2t/2^n$ (here there are two values I_{t+1}, I'_{t+1} to consider; note that they cannot be equal to each other). Finally, arguing as before, for $k > t$ we have $\Pr[\mathsf{Coll}(k) \mid \overline{\mathsf{Coll}(k-1)}] = (k+1)/2^n$. Using Equation (4.7), we thus have

$$\begin{aligned}\Pr[\mathsf{Coll}_{i,j}] &\leq 2^{-n} \cdot \left(\sum_{k=1}^{t-1} k + 2t + \sum_{k=t+1}^{2\ell-2} (k+1) \right) \\ &= 2^{-n} \cdot \sum_{k=2}^{2\ell-1} k \;\; = \;\; 2^{-n} \cdot (2\ell+1) \cdot (\ell-1) \;\; < \;\; 2\ell^2 \cdot 2^{-n}.\end{aligned}$$

From Equation (4.6) we get $\Pr[\mathsf{Coll}] < q^2\ell^2 \cdot 2^{-n} = \delta$. Finally, using Equation (4.5) we see that

$$\begin{aligned}\Pr\left[\forall i : \mathsf{CBC}_g(X_i) = t_i\right] &\geq \Pr\left[\forall i : \mathsf{CBC}_g(X_i) = t_i \mid \overline{\mathsf{Coll}}\right] \cdot \Pr[\overline{\mathsf{Coll}}] \\ &= 2^{-nq} \cdot \Pr[\overline{\mathsf{Coll}}] \;\; \geq (1-\delta) \cdot 2^{-nq},\end{aligned}$$

as claimed. ∎

We now show that smoothness implies the theorem. Assume without loss of generality that D always makes q (distinct) queries, each containing at most ℓ blocks. D may choose its queries adaptively (i.e., depending on the answers to previous queries), but the set of D's queries must be prefix-free.

For distinct $X_1, \ldots, X_q \in (\{0,1\}^n)^*$ and arbitrary $t_1, \ldots, t_q \in \{0,1\}^n$, define $\alpha(X_1, \ldots, X_q; t_1, \ldots, t_q)$ to be 1 if and only if D outputs 1 when making queries $X_1, \ldots, X_q$ and getting responses $t_1, \ldots, t_q$. (If, say, D does not make query X_1 as its first query, then $\alpha(X_1, \ldots; \ldots) = 0$.) Letting $\vec{X} = (X_1, \ldots, X_q)$ and $\vec{t} = (t_1, \ldots, t_q)$, we then have

$$\begin{aligned}\Pr[D^{\mathsf{CBC}_g(\cdot)}(1^n) = 1] &= \sum_{\vec{X} \text{ prefix-free; } \vec{t}} \alpha(\vec{X}, \vec{t}) \cdot \Pr[\forall i : \mathsf{CBC}_g(X_i) = t_i] \\ &\geq \sum_{\vec{X} \text{ prefix-free; } \vec{t}} \alpha(\vec{X}, \vec{t}) \cdot (1-\delta) \cdot \Pr[\forall i : f(X_i) = t_i] \\ &= (1-\delta) \cdot \Pr[D^{f(\cdot)}(1^n) = 1],\end{aligned}$$

where, above, g is chosen uniformly from Func_n, and f is chosen uniformly from the set of functions mapping $(\{0,1\}^n)^*$ to $\{0,1\}^n$. This implies

$$\Pr[D^{f(\cdot)}(1^n) = 1] - \Pr[D^{\mathsf{CBC}_g(\cdot)}(1^n) = 1] \leq \delta \cdot \Pr[D^{f(\cdot)}(1^n) = 1] \leq \delta.$$

A symmetric argument for when D outputs 0 completes the proof. ∎

4.5 Authenticated Encryption

In Chapter 3, we studied how it is possible to obtain *secrecy* in the private-key setting using encryption. In this chapter, we have shown how to ensure *integrity* using message authentication codes. One might naturally want to achieve both goals simultaneously, and this is the problem we turn to now.

It is best practice to *always ensure secrecy and integrity by default* in the private-key setting. Indeed, in many applications where secrecy is required it turns out that integrity is essential also. Moreover, a lack of integrity can sometimes lead to a breach of secrecy.

4.5.1 Definitions

We begin, as usual, by defining precisely what we wish to achieve. At an abstract level, our goal is to realize an "ideally secure" communication channel that provides both secrecy and integrity. Pursuing a definition of this sort is beyond the scope of this book. Instead, we provide a simpler set of definitions that treat secrecy and integrity separately. These definitions and our subsequent analysis suffice for understanding the key issues at hand. (We caution the reader, however, that—in contrast to encryption and message authentication codes—the field has not yet settled on standard terminology and definitions for authenticated encryption.)

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme. As mentioned already, we define security by separately defining secrecy and integrity. The notion of secrecy we consider is one we have seen before: we require that Π be secure against chosen-ciphertext attacks, i.e., that it be CCA-secure. (Refer to Section 3.7 for a discussion and definition of CCA-security.) We are concerned about chosen-ciphertext attacks here because we are explicitly considering an active adversary who can modify the data sent from one honest party to the other. Our notion of integrity will be essentially that of existential unforgeability under an adaptive chosen-message attack. Since Π does not satisfy the syntax of a message authentication code, however, we introduce a definition specific to this case. Consider the following experiment defined for a private-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, adversary $\mathcal{A}$, and value n for the security parameter:

The unforgeable encryption experiment $\mathsf{Enc\text{-}Forge}_{\mathcal{A},\Pi}(n)$**:**

1. *Run* $\mathsf{Gen}(1^n)$ *to obtain a key* k.
2. *The adversary* $\mathcal{A}$ *is given input* 1^n *and access to an encryption oracle* $\mathsf{Enc}_k(\cdot)$. *The adversary outputs a ciphertext* c.
3. *Let* $m := \mathsf{Dec}_k(c)$, *and let* $\mathcal{Q}$ *denote the set of all queries that* $\mathcal{A}$ *asked its encryption oracle. The output of the experiment is* 1 *if and only if (1)* $m \neq \perp$ *and (2)* $m \notin \mathcal{Q}$.

DEFINITION 4.16 *A private-key encryption scheme* Π *is* unforgeable *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that:*

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

Paralleling our discussion about verification queries following Definition 4.2, here one could also consider a stronger definition in which $\mathcal{A}$ is additionally given access to a decryption oracle. One can verify that the secure construction we present below also satisfies that stronger definition.

We now define a (secure) *authenticated encryption* scheme.

DEFINITION 4.17 *A private-key encryption scheme is an* authenticated encryption scheme *if it is CCA-secure and unforgeable.*

4.5.2 Generic Constructions

It may be tempting to think that any reasonable combination of a secure encryption scheme and a secure message authentication code should result in an authenticated encryption scheme. In this section we show that this is not the case. This demonstrates that even secure cryptographic tools can be combined in such a way that the result is insecure, and highlights once again the importance of definitions and proofs of security. On the positive side, we show how encryption and message authentication can be combined properly to achieve joint secrecy and integrity.

Throughout, let $\Pi_E = (\mathsf{Enc}, \mathsf{Dec})$ be a CPA-secure encryption scheme and let $\Pi_M = (\mathsf{Mac}, \mathsf{Vrfy})$ denote a message authentication code, where key generation in both schemes simply involves choosing a uniform n-bit key. There are three natural approaches to combining encryption and message authentication using independent keys[4] k_E and k_M for Π_E and Π_M, respectively:

1. *Encrypt-and-authenticate:* In this method, encryption and message authentication are computed independently in parallel. That is, given a plaintext message m, the sender transmits the ciphertext $\langle c, t\rangle$ where:

   $$c \leftarrow \mathsf{Enc}_{k_E}(m) \quad \text{and} \quad t \leftarrow \mathsf{Mac}_{k_M}(m).$$

   The receiver decrypts c to recover m; assuming no error occurred, it then verifies the tag t. If $\mathsf{Vrfy}_{k_M}(m, t) = 1$, the receiver outputs m; otherwise, it outputs an error.

[4] *Independent* cryptographic keys should always be used when different schemes are combined. We return to this point at the end of this section.

2. *Authenticate-then-encrypt:* Here a MAC tag t is first computed, and then the message and tag are encrypted together. That is, given a message m, the sender transmits the ciphertext c computed as:

$$t \leftarrow \mathsf{Mac}_{k_M}(m) \;\text{ and }\; c \leftarrow \mathsf{Enc}_{k_E}(m \| t).$$

   The receiver decrypts c to obtain $m \| t$; assuming no error occurs, it then verifies the tag t. As before, if $\mathsf{Vrfy}_{k_M}(m, t) = 1$ the receiver outputs m; otherwise, it outputs an error.

3. *Encrypt-then-authenticate:* In this case, the message m is first encrypted and then a MAC tag is computed over the result. That is, the ciphertext is the pair $\langle c, t \rangle$ where:

$$c \leftarrow \mathsf{Enc}_{k_E}(m) \;\text{ and }\; t \leftarrow \mathsf{Mac}_{k_M}(c).$$

   (See also Construction 4.18.) If $\mathsf{Vrfy}_{k_M}(c, t) = 1$, then the receiver decrypts c and outputs the result; otherwise, it outputs an error.

We analyze each of the above approaches when they are instantiated with "generic" secure components, i.e., an *arbitrary* CPA-secure encryption scheme and an *arbitrary* (strongly) secure MAC. We want an approach that provides joint secrecy and integrity when using *any* (secure) components, and we will therefore reject as "unsafe" any approach for which there exists even a single counterexample of a secure encryption scheme/MAC for which the combination is insecure. This "all-or-nothing" approach reduces the likelihood of implementation flaws. Specifically, an authenticated encryption scheme might be implemented by making calls to an "encryption subroutine" and a "message authentication subroutine," and the implementation of those subroutines may be changed at some later point in time. (This commonly occurs when cryptographic libraries are updated, or when standards are modified.) Implementing an approach whose security depends on how its components are implemented (rather than on the security they provide) is therefore dangerous.

We stress that if an approach is rejected this does not mean that it is insecure for all possible instantiations of the components; it does, however, mean that any instantiation of the approach must be analyzed and proven secure before it is used.

Encrypt-and-authenticate. Recall that in this approach encryption and message authentication are carried out independently. Given a message m, the transmitted value is $\langle c, t \rangle$ where

$$c \leftarrow \mathsf{Enc}_{k_E}(m) \;\text{ and }\; t \leftarrow \mathsf{Mac}_{k_M}(m).$$

This approach may not achieve even the most basic level of secrecy. To see this, note that a secure MAC does not guarantee *any* secrecy and so it is possible for the tag $\mathsf{Mac}_{k_M}(m)$ to leak information about m to an eavesdropper.

(As a trivial example, consider a secure MAC where the first bit of the tag is always equal to the first bit of the message.) So the encrypt-and-authenticate approach may yield a scheme that does not even have indistinguishable encryptions in the presence of an eavesdropper.

In fact, the encrypt-and-authenticate approach is likely to be insecure against chosen-plaintext attacks even when instantiated with standard components (unlike the contrived counterexample in the previous paragraph). In particular, if a *deterministic* MAC like CBC-MAC is used, then the tag computed on a message (for some fixed key k_M) is the same every time. This allows an eavesdropper to identify when the same message is sent twice, and so the scheme is not CPA-secure. Most MACs used in practice are deterministic, so this represents a real concern.

Authenticate-then-encrypt. Here, a MAC tag $t \leftarrow \mathsf{Mac}_{k_M}(m)$ is first computed; then $m\|t$ is encrypted and the resulting value $\mathsf{Enc}_{k_E}(m\|t)$ is transmitted. We show that this combination also does not necessarily yield an authenticated encryption scheme.

Actually, we have already encountered a CPA-secure encryption scheme for which this approach is insecure: the CBC-mode-with-padding scheme discussed in Section 3.7.2. (We assume in what follows that the reader is familiar with that section.) Recall that this scheme works by first padding the plaintext (which in our case will be $m\|t$) in a specific way so the result is a multiple of the block length, and then encrypting the result using CBC mode. During decryption, if an error in the padding is detected after performing the CBC-mode decryption, then a "bad padding" error is returned. With regard to authenticate-then-encrypt, this means there are now *two* sources of potential decryption failure: the padding may be incorrect, or the MAC tag may not verify. Schematically, the decryption algorithm Dec' in the combined scheme works as follows:

$\mathsf{Dec}'_{k_E,k_M}(c)$**:**

1. Compute $\tilde{m} := \mathsf{Dec}_{k_E}(c)$. If an error in the padding is detected, return "bad padding" and stop.
2. Parse $\tilde{m}$ as $m\|t$. If $\mathsf{Vrfy}_{k_M}(m,t) = 1$ return m; else, output "authentication failure."

Assuming the attacker can distinguish between the two error messages, the attacker can apply the same chosen-ciphertext attack described in Section 3.7.2 to the above scheme to recover the entire original plaintext from a given ciphertext. (This is due to the fact that the padding-oracle attack shown in Section 3.7.2 relies only on the ability to learn whether or not there was a padding error, something that is revealed by this scheme.) This type of attack has been carried out successfully in the real world in various settings, e.g., in configurations of IPsec that use authenticate-then-encrypt.

One way to fix the above scheme would be to ensure that only a *single* error message is returned, regardless of the source of decryption failure. This is an unsatisfying solution for several reasons: (1) there may be legitimate reasons (e.g., usability, debugging) to have multiple error messages; (2) forcing the error messages to be the same means that the combination is no longer truly generic, i.e., it requires the implementer of the authenticate-then-encrypt approach to be aware of what error messages are returned by the underlying CPA-secure encryption scheme; (3) most of all, it is extraordinarily hard to ensure that the different errors cannot be distinguished since, e.g., even a difference in the time to return each of these errors may be used by an adversary to distinguish between them (cf. our earlier discussion of timing attacks at the end of Section 4.2). Some versions of SSL tried using only a single error message in conjunction with an authenticate-then-encrypt approach, but a padding-oracle attack was still successfully carried out using timing information of this sort. We conclude that authenticate-then-encrypt does not provide authenticated encryption in general, and should not be used.

Encrypt-then-authenticate. In this approach, the message is first encrypted and then a MAC is computed over the result. That is, the message is the pair $\langle c, t\rangle$ where

$$c \leftarrow \mathsf{Enc}_{k_E}(m) \quad \text{and} \quad t \leftarrow \mathsf{Mac}_{k_M}(c).$$

Decryption of $\langle c, t\rangle$ is done by outputting $\perp$ if $\mathsf{Vrfy}_{k_M}(c, t) \neq 1$, and otherwise outputting $\mathsf{Dec}_{k_E}(c)$. See Construction 4.18 for a formal description.

CONSTRUCTION 4.18

Let $\Pi_E = (\mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme and let $\Pi_M = (\mathsf{Mac}, \mathsf{Vrfy})$ be a message authentication code, where in each case key generation is done by simply choosing a uniform n-bit key. Define a private-key encryption scheme $(\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ as follows:

- Gen': on input 1^n, choose independent, uniform $k_E, k_M \in \{0,1\}^n$ and output the key (k_E, k_M).
- Enc': on input a key (k_E, k_M) and a plaintext message m, compute $c \leftarrow \mathsf{Enc}_{k_E}(m)$ and $t \leftarrow \mathsf{Mac}_{k_M}(c)$. Output the ciphertext $\langle c, t\rangle$.
- Dec': on input a key (k_E, k_M) and a ciphertext $\langle c, t\rangle$, first check whether $\mathsf{Vrfy}_{k_M}(c, t) \stackrel{?}{=} 1$. If yes, then output $\mathsf{Dec}_{k_E}(c)$; if no, then output $\perp$.

A generic construction of an authenticated encryption scheme.

This approach *is* sound, as long as the MAC is *strongly secure*, as in Definition 4.3. As intuition for the security of this approach, say a ciphertext $\langle c, t\rangle$ is *valid* if t is a valid MAC tag on c. Strong security of the MAC ensures that an

adversary will be unable to generate *any* valid ciphertext that it did not receive from its encryption oracle. This immediately implies that Construction 4.18 is unforgeable. As for CCA-security, the MAC computed over the ciphertext has the effect of rendering the decryption oracle useless since for every ciphertext $\langle c, t\rangle$ the adversary submits to its decryption oracle, the adversary either already knows the decryption (if it received $\langle c, t\rangle$ from its own encryption oracle) or else can expect the result to be an error (since the adversary cannot generate any new, valid ciphertexts). This means that CCA-security of the combined scheme reduces to the CPA-security of Π_E. Observe also that the MAC is verified before decryption takes place; thus, MAC verification cannot leak anything about the plaintext (in contrast to the padding-oracle attack we saw for the authenticate-then-encrypt approach). We now formalize the above arguments.

THEOREM 4.19 *Let Π_E be a CPA-secure private-key encryption scheme, and let Π_M be a strongly secure message authentication code. Then Construction 4.18 is an authenticated encryption scheme.*

PROOF Let Π' denote the scheme resulting from Construction 4.18. We need to show that Π' is unforgeable, and that it is CCA-secure. Following the intuition given above, say a ciphertext $\langle c, t\rangle$ is *valid* (with respect to some fixed secret key (k_E, k_M)) if $\mathsf{Vrfy}_{k_M}(c, t) = 1$. We show that strong security of Π_M implies that (except with negligible probability) any "new" ciphertexts the adversary submits to the decryption oracle will be invalid. As discussed already, this immediately implies unforgeability. (In fact, it is stronger than unforgeability.) This fact also renders the decryption oracle useless, and means that CCA-security of $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ reduces to the CPA-security of Π_E.

In more detail, let $\mathcal{A}$ be a probabilistic polynomial-time adversary attacking Construction 4.18 in a chosen-ciphertext attack (cf. Definition 3.33). Say a ciphertext $\langle c, t\rangle$ is *new* if $\mathcal{A}$ did not receive $\langle c, t\rangle$ from its encryption oracle or as the challenge ciphertext. Let $\mathsf{ValidQuery}$ be the event that $\mathcal{A}$ submits a new ciphertext $\langle c, t\rangle$ to its decryption oracle which is valid, i.e., for which $\mathsf{Vrfy}_{k_M}(c, t) = 1$. We prove:

CLAIM 4.20 $\Pr[\mathsf{ValidQuery}]$ *is negligible.*

PROOF Intuitively, this is due to the fact that if $\mathsf{ValidQuery}$ occurs then the adversary has forged a new, valid pair (c, t) in the $\mathsf{Mac\text{-}sforge}$ experiment. Formally, let $q(\cdot)$ be a polynomial upper bound on the number of decryption-oracle queries made by $\mathcal{A}$, and consider the following adversary $\mathcal{A}_M$ attacking the message authentication code Π_M (i.e., running in experiment $\mathsf{Mac\text{-}sforge}_{\mathcal{A}_M, \Pi_M}(n)$):

Adversary $\mathcal{A}_M$:
$\mathcal{A}_M$ is given input 1^n and has access to a MAC oracle $\mathsf{Mac}_{k_M}(\cdot)$.

1. Choose uniform $k_E \in \{0,1\}^n$ and $i \in \{1,\ldots,q(n)\}$.
2. Run $\mathcal{A}$ on input 1^n. When $\mathcal{A}$ makes an encryption-oracle query for the message m, answer it as follows:
   (i) Compute $c \leftarrow \mathsf{Enc}_{k_E}(m)$.
   (ii) Query c to the MAC oracle and receive t in response. Return $\langle c,t\rangle$ to $\mathcal{A}$.

   The challenge ciphertext is prepared in the exact same way (with a uniform bit $b \in \{0,1\}$ chosen to select the message m_b that gets encrypted).

   When $\mathcal{A}$ makes a decryption-oracle query for the ciphertext $\langle c,t\rangle$, answer it as follows: If this is the ith decryption-oracle query, output (c,t). Otherwise:
   (i) If $\langle c,t\rangle$ was a response to a previous encryption-oracle query for a message m, return m.
   (ii) Otherwise, return $\perp$.

In essence, $\mathcal{A}_M$ is "guessing" that the ith decryption-oracle query of $\mathcal{A}$ will be the first new, valid query $\mathcal{A}$ makes. In that case, $\mathcal{A}_M$ outputs a valid forgery on a message c that it had never previously submitted to its own MAC oracle.

Clearly $\mathcal{A}_M$ runs in probabilistic polynomial time. We now analyze the probability that $\mathcal{A}_M$ produces a good forgery. The key point is that the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}_M$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$ *until event* $\mathsf{ValidQuery}$ *occurs*. To see this, note that the encryption-oracle queries of $\mathcal{A}$ (as well as computation of the challenge ciphertext) are simulated perfectly by $\mathcal{A}_M$. As for the decryption-oracle queries of $\mathcal{A}$, until $\mathsf{ValidQuery}$ occurs these are all simulated properly. In case (i) this is obvious. As for case (ii), if the ciphertext $\langle c,t\rangle$ submitted to the decryption oracle is new, then as long as $\mathsf{ValidQuery}$ has not yet occurred the correct answer to the decryption-oracle query is indeed $\perp$. (Note that case (i) is exactly the case that $\langle c,t\rangle$ is not new, and case (ii) is exactly the case that $\langle c,t\rangle$ is new.) Recall that $\mathcal{A}$ is disallowed from submitting the challenge ciphertext to the decryption oracle.

Because the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}_M$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$ until event $\mathsf{ValidQuery}$ occurs, the probability of event $\mathsf{ValidQuery}$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}_M,\Pi_M}(n)$ is the same as the probability of that event in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$.

If $\mathcal{A}_M$ correctly guesses the first index i when $\mathsf{ValidQuery}$ occurs, then $\mathcal{A}_M$ outputs (c,t) for which $\mathsf{Vrfy}_{k_M}(c,t) = 1$ (since $\langle c,t\rangle$ is valid) and for which it was never given tag t in response to the query $\mathsf{Mac}_{k_M}(c)$ (since $\langle c,t\rangle$ is new). In this case, then, $\mathcal{A}_M$ succeeds in experiment $\mathsf{Mac\text{-}sforge}_{\mathcal{A}_M,\Pi_M}(n)$.

The probability that $\mathcal{A}_M$ guesses i correctly is $1/q(n)$. Therefore

$$\Pr[\mathsf{Mac\text{-}sforge}_{\mathcal{A}_M,\Pi_M}(n) = 1] \geq \Pr[\mathsf{ValidQuery}]/q(n).$$

Since Π_M is a strongly secure MAC and q is polynomial, we conclude that $\Pr[\mathsf{ValidQuery}]$ is negligible. ■

We use Claim 4.20 to prove security of Π'. The easier case is to prove that Π' is unforgeable. This follows immediately from the claim, and so we just provide informal reasoning rather than a formal proof. Observe first that the adversary $\mathcal{A}'$ in the unforgeable encryption experiment is a restricted version of the adversary in the chosen-ciphertext experiment (in the former, the adversary only has access to an encryption oracle). When $\mathcal{A}'$ outputs a ciphertext $\langle c, t\rangle$ at the end of its experiment, it "succeeds" only if $\langle c, t\rangle$ is valid and new. But the previous claim shows precisely that the probability of such an event is negligible.

It is slightly more involved to prove that Π' is CCA-secure. Let $\mathcal{A}$ again be a probabilistic polynomial-time adversary attacking Π' in a chosen-ciphertext attack. We have

$$\begin{aligned}\Pr[&\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1] \\ &\leq \Pr[\mathsf{ValidQuery}] + \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}]. \qquad (4.8)\end{aligned}$$

We have already shown that $\Pr[\mathsf{ValidQuery}]$ is negligible. The following claim thus completes the proof of the theorem.

CLAIM 4.21 *There exists a negligible function* negl *such that*

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

To prove this claim, we rely on CPA-security of Π_E. Consider the following adversary $\mathcal{A}_E$ attacking Π_E in a chosen-plaintext attack:

Adversary $\mathcal{A}_E$:
$\mathcal{A}_E$ is given input 1^n and has access to $\mathsf{Enc}_{k_E}(\cdot)$.

1. Choose uniform $k_M \in \{0,1\}^n$.
2. Run $\mathcal{A}$ on input 1^n. When $\mathcal{A}$ makes an encryption-oracle query for the message m, answer it as follows:
   (i) Query m to $\mathsf{Enc}_{k_E}(\cdot)$ and receive c in response.
   (ii) Compute $t \leftarrow \mathsf{Mac}_{k_M}(c)$ and return $\langle c, t\rangle$ to $\mathcal{A}$.

   When $\mathcal{A}$ makes a decryption-oracle query for the ciphertext $\langle c, t\rangle$, answer it as follows:

- If $\langle c, t\rangle$ was a response to a previous encryption-oracle query for a message m, return m. Otherwise, return $\perp$.

3. When $\mathcal{A}$ outputs messages (m_0, m_1), output these same messages and receive a challenge ciphertext c in response. Compute $t \leftarrow \mathsf{Mac}_{k_M}(c)$, and return $\langle c, t\rangle$ as the challenge ciphertext for $\mathcal{A}$. Continue answering $\mathcal{A}$'s oracle queries as above.
4. Output the same bit b' that is output by $\mathcal{A}$.

Notice that $\mathcal{A}_E$ does not need a decryption oracle because it simply assumes that any decryption query by $\mathcal{A}$ that was not the result of a previous encryption-oracle query is invalid.

Clearly, $\mathcal{A}_E$ runs in probabilistic polynomial time. Furthermore, the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}_E$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n)$ *as long as event* $\mathsf{ValidQuery}$ *never occurs*. Therefore, the probability that $\mathcal{A}_E$ succeeds when $\mathsf{ValidQuery}$ does not occur is the same as the probability that $\mathcal{A}$ succeeds when $\mathsf{ValidQuery}$ does not occur; i.e.,

$$\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}] = \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}],$$

implying that

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1] &\geq \Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}] \\ &= \Pr[\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A},\Pi'}(n) = 1 \wedge \overline{\mathsf{ValidQuery}}].\end{aligned}$$

Since Π_E is CPA-secure, there exists a negligible function negl such that $\Pr[\mathsf{PrivK}^{\mathsf{cpa}}_{\mathcal{A}_E,\Pi_E}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$. This proves the claim. ∎

The need for independent keys. We conclude this section by stressing a basic principle of cryptography: *different instances of cryptographic primitives should always use independent keys.* To illustrate this here, consider what can happen to the encrypt-then-authenticate methodology when the same key k is used for both encryption and authentication. Let F be a strong pseudorandom permutation. It follows that F^{-1} is a strong pseudorandom permutation also. Define $\mathsf{Enc}_k(m) = F_k(m\|r)$ for $m \in \{0,1\}^{n/2}$ and a uniform $r \in \{0,1\}^{n/2}$, and define $\mathsf{Mac}_k(c) = F_k^{-1}(c)$. It can be shown that this encryption scheme is CPA-secure (in fact, it is even CCA-secure; see Exercise 4.25), and we know that the given message authentication code is a secure MAC. However, the encrypt-then-authenticate combination using the same key k as applied to the message m yields:

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

and the message m is revealed in the clear! This does not in any way contradict Theorem 4.19, since Construction 4.18 expressly requires that k_M, k_E are chosen (uniformly and) independently. We encourage the reader to examine where this independence is used in the proof of Theorem 4.19.

4.5.3 Secure Communication Sessions

We briefly describe the application of authenticated encryption to the setting of two parties who wish to communicate "securely"—namely, with joint secrecy and integrity—over the course of a communication session. (For the purposes of this section, a *communication session* is simply a period of time during which the communicating parties maintain state.) In our treatment here we are deliberately informal; a formal definition is quite involved, and this topic arguably lies more in the area of network security than cryptography.

Let $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ be an authenticated encryption scheme. Consider two parties A and B who share a key k and wish to use this key to secure their communication over the course of a session. The obvious thing to do here is to use Π: Whenever, say, A wants to transmit a message m to B, it computes $c \leftarrow \mathsf{Enc}_k(m)$ and sends c to B; in turn, B decrypts c to recover the result (ignoring the result if decryption returns $\perp$). Likewise, the same procedure is followed when B wants to send a message to A. This simple approach, however, does not suffice, as there are various potential attacks:

Re-ordering attack An attacker can swap the order of messages. For example, if A transmits c_1 (an encryption of m_1) and subsequently transmits c_2 (an encryption of m_2), an attacker who has some control over the network can deliver c_2 before c_1 and thus cause B to output the messages in the wrong order. This causes a mismatch between the two parties' views of their communication session.

Replay attack An attacker can *replay* a (valid) ciphertext c sent previously by one of the parties. Again, this causes a mismatch between what is sent by one party and received by the other.

Reflection attack An attacker can take a ciphertext c sent from A to B and send it back to A. This again can cause a mismatch between the two parties' transcripts of their communication session: A may output a message m, even though B never sent such a message.

Fortunately, the above attacks are easy to prevent using *counters* to address the first two and a *directionality bit* to prevent the third.[5] We describe these in tandem. Each party maintains two counters $\mathsf{ctr}_{A,B}$ and $\mathsf{ctr}_{B,A}$ keeping track of the number of messages sent from A to B (resp., B to A) during the session. These counters are initialized to 0 and incremented each time a party sends or receives a (valid) message. The parties also agree on a bit $b_{A,B}$, and define $b_{B,A}$ to be its complement. (One way to do this is to set $b_{A,B} = 0$ iff the identity of A is lexicographically smaller than the identity of B.)

[5] In practice, the issue of directionality is often solved by simply having separate keys for each direction (i.e., the parties use a key k_A for messages sent from A to B, and a different key k_B for messages sent from B to A).

When A wants to transmit a message m to B, she computes the ciphertext $c \leftarrow \mathsf{Enc}_k(b_{A,B} \| \mathsf{ctr}_{A,B} \| m)$ and sends c; she then increments $\mathsf{ctr}_{A,B}$. Upon receiving c, party B decrypts; if the result is $\perp$, he immediately rejects. Otherwise, he parses the decrypted message as $b \| \mathsf{ctr} \| m$. If $b = b_{A,B}$ and $\mathsf{ctr} = \mathsf{ctr}_{A,B}$, then B outputs m and increments $\mathsf{ctr}_{A,B}$; otherwise, B rejects. The above steps, *mutatis mutandis*, are applied when B sends a message to A.

We remark that since the parties are anyway maintaining state (namely, the counters $\mathsf{ctr}_{A,B}$ and $\mathsf{ctr}_{B,A}$), the parties could easily use a *stateful* authenticated encryption scheme Π.

4.5.4 CCA-Secure Encryption

It follows directly from the definition that any authenticated encryption scheme is also secure against chosen-ciphertext attacks. Can there be CCA-secure private-key encryption schemes that are *not* unforgeable? Indeed, there are; see Exercise 4.25.

One can imagine applications where CCA-security is needed but authenticated encryption is not. One example might be when private-key encryption is used for *key transport*. As a concrete example, say a server gives a tamper-proof hardware token to a user, where embedded in the token is a long-term key k. The server can upload a fresh, short-term key k' to this token by giving the user $\mathsf{Enc}_k(k')$; the user is supposed to give this ciphertext to the token, which will decrypt it and use k' for the next time period. A chosen-ciphertext attack in this setting could allow the user to learn k', something the user is not supposed to be able to do. (Note that here a padding-oracle attack, which only relies on the user's ability to determine whether a decryption failure occurs, could potentially be carried out rather easily.) On the other hand, not much harm is done if the user can generate a "valid" ciphertext that causes the token to use an arbitrary (unrelated) key k'' for the next time period. (Of course, this depends on what the token does with this short-term key.)

Notwithstanding the above, for private-key encryption most "natural" constructions of CCA-secure schemes that we know anyway satisfy the stronger definition of authenticated encryption. Put differently, there is no real reason to ever use a CCA-secure scheme that is *not* an authenticated encryption scheme, simply because we don't really have any constructions satisfying the former that are more efficient than constructions achieving the latter.

From a *conceptual* point of view, however, it is important to keep the notions of CCA-security and authenticated encryption distinct. With regard to CCA-security we are not interested in message integrity *per se*; rather, we wish to ensure privacy even against an active adversary who can interfere with the communication as it goes from sender to receiver. In contrast, with regard to authenticated encryption we are interested in the twin goals of secrecy and integrity. We stress this here because in the public-key setting that we study later in the book, the difference between authenticated encryption and CCA-security is more pronounced.

4.6 *Information-Theoretic MACs

In previous sections we have explored message authentication codes with *computational* security, i.e., where bounds on the attacker's running time are assumed. Recalling the results of Chapter 2, it is natural to ask whether message authentication in the presence of an *unbounded* adversary is possible. In this section, we show under which conditions *information-theoretic* (as opposed to computational) security is attainable.

A first observation is that it is impossible to achieve "perfect" security in this context: namely, we cannot hope to have a message authentication code for which the probability that an adversary outputs a valid tag on a previously unauthenticated message is 0. The reason is that an adversary can simply guess a valid tag t on any message and the guess will be correct with probability (at least) $1/2^{|t|}$, where $|t|$ denotes the tag length of the scheme.

The above example tells us what we *can* hope to achieve: a MAC with tags of length $|t|$ where the probability of forgery is at most $1/2^{|t|}$, even for unbounded adversaries. We will see that this is achievable, but only under restrictions on how many messages are authenticated by the honest parties.

We first define information-theoretic security for message authentication codes. A starting point is to take experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$ that is used to define security for computationally secure MACs (cf. Definition 4.2), but to drop the security parameter n and require that $\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi} = 1]$ should be "small" for *all* adversaries $\mathcal{A}$ (and not just adversaries running in polynomial time). As mentioned above (and as will be proved formally in Section 4.6.2), however, such a definition is impossible to achieve. Rather, information-theoretic security can be achieved only if we place some bound on the number of messages authenticated by the honest parties. We look here at the most basic setting, where the parties authenticate just a *single* message. We refer to this as *one-time message authentication*. The following experiment modifies $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$ following the above discussion:

The one-time message authentication experiment $\mathsf{Mac\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi}$:

1. *A key k is generated by running* Gen.
2. *The adversary $\mathcal{A}$ outputs a message m', and is given in return a tag $t' \leftarrow \mathsf{Mac}_k(m')$.*
3. *$\mathcal{A}$ outputs (m, t).*
4. *The output of the experiment is defined to be* 1 *if and only if (1) $\mathsf{Vrfy}_k(m, t) = 1$ and (2) $m \neq m'$.*

DEFINITION 4.22 *A message authentication code $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ is* one-time ε-secure*, or just ε-secure, if for all (even unbounded) adversaries $\mathcal{A}$:*

$$\Pr[\mathsf{Mac\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi} = 1] \leq \varepsilon.$$

4.6.1 Constructing Information-Theoretic MACs

In this section we show how to build an ε-secure MAC based on any *strongly universal function*.[6] We then show a simple construction of the latter.

Let $h : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ be a keyed function whose first input is a key $k \in \mathcal{K}$ and whose second input is taken from some domain $\mathcal{M}$. As usual, we write $h_k(m)$ instead of $h(k, m)$. Then h is *strongly universal* (or *pairwise-independent*) if for any two distinct inputs m, m' the values $h_k(m)$ and $h_k(m')$ are uniformly and independently distributed in $\mathcal{T}$ when k is a uniform key. This is equivalent to saying that the probability that $h_k(m), h_k(m')$ take on any particular values t, t' is exactly $1/|\mathcal{T}|^2$. That is:

DEFINITION 4.23 *A function $h : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ is* strongly universal *if for all distinct $m, m' \in \mathcal{M}$ and all $t, t' \in \mathcal{T}$ it holds that*

$$\Pr[h_k(m) = t \wedge h_k(m') = t'] = \frac{1}{|\mathcal{T}|^2},$$

where the probability is taken over uniform choice of $k \in \mathcal{K}$.

The above should motivate the construction of a one-time message authentication code from any strongly universal function h. The tag t on a message m is obtained by computing $h_k(m)$, where the key k is uniform; see Construction 4.24. Intuitively, even after an adversary observes the tag $t' := h_k(m')$ for any message m', the correct tag $h_k(m)$ for any *other* message m is still uniformly distributed in $\mathcal{T}$ from the adversary's point of view. Thus, the adversary can do nothing more than blindly guess the tag, and this guess will be correct only with probability $1/|\mathcal{T}|$.

CONSTRUCTION 4.24

Let $h : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ be a strongly universal function. Define a MAC for messages in $\mathcal{M}$ as follows:

- Gen: choose uniform $k \in \mathcal{K}$ and output it as the key.
- Mac: on input a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, output the tag $t := h_k(m)$.
- Vrfy: on input a key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$, and a tag $t \in \mathcal{T}$, output 1 if and only if $t \stackrel{?}{=} h_k(m)$. (If $m \notin \mathcal{M}$, then output 0.)

A MAC from any strongly universal function.

[6] These are often called strongly universal *hash* functions, but in cryptographic contexts the term "hash" has another meaning that we will see later in the book.

The above construction can be viewed as analogous to Construction 4.5. This is because a strongly universal function h is identical to a random function, as long as it is only evaluated twice.

THEOREM 4.25 *Let $h : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ be a strongly universal function. Then Construction 4.24 is a $1/|\mathcal{T}|$-secure MAC for messages in $\mathcal{M}$.*

PROOF Let $\mathcal{A}$ be an adversary. As usual in the information-theoretic setting, we may assume $\mathcal{A}$ is deterministic without loss of generality. So the message m' that $\mathcal{A}$ outputs at the outset of the experiment is fixed. Furthermore, the pair (m, t) that $\mathcal{A}$ outputs at the end of the experiment is a deterministic function of the tag t' on m' that $\mathcal{A}$ receives. We thus have

$$\begin{aligned}\Pr[\mathsf{Mac\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi} = 1] &= \sum_{t' \in \mathcal{T}} \Pr[\mathsf{Mac\text{-}forge}^{\mathsf{1\text{-}time}}_{\mathcal{A},\Pi} = 1 \ \wedge \ h_k(m') = t'] \\ &= \sum_{\substack{t' \in \mathcal{T} \\ (m,t) := \mathcal{A}(t')}} \Pr[h_k(m) = t \ \wedge \ h_k(m') = t'] \\ &= \sum_{\substack{t' \in \mathcal{T} \\ (m,t) := \mathcal{A}(t')}} \frac{1}{|\mathcal{T}|^2} \ = \ \frac{1}{|\mathcal{T}|}.\end{aligned}$$

This proves the theorem. ■

We now turn to a classical construction of a strongly universal function. We assume some basic knowledge about arithmetic modulo a prime number; readers may refer to Sections 8.1.1 and 8.1.2 for the necessary background. Fix some prime p, and let $\mathbb{Z}_p \stackrel{\text{def}}{=} \{0, \ldots, p-1\}$. We take as our message space $\mathcal{M} = \mathbb{Z}_p$; the space of possible tags will also be $\mathcal{T} = \mathbb{Z}_p$. A key (a, b) consists of a pair of elements from $\mathbb{Z}_p$; thus, $\mathcal{K} = \mathbb{Z}_p \times \mathbb{Z}_p$. Define h as

$$h_{a,b}(m) \stackrel{\text{def}}{=} [a \cdot m + b \bmod p],$$

where the notation $[X \bmod p]$ refers to the reduction of the integer X modulo p (and so $[X \bmod p] \in \mathbb{Z}_p$ always).

THEOREM 4.26 *For any prime p, the function h is strongly universal.*

PROOF Fix any distinct $m, m' \in \mathbb{Z}_p$ and any $t, t' \in \mathbb{Z}_p$. For which keys (a, b) does it hold that both $h_{a,b}(m) = t$ and $h_{a,b}(m') = t'$? This holds only if

$$a \cdot m + b = t \bmod p \quad \text{and} \quad a \cdot m' + b = t' \bmod p.$$

We thus have two linear equations in the two unknowns a, b. These two equations are both satisfied exactly when $a = [(t-t') \cdot (m-m')^{-1} \bmod p]$ and $b = [t - a \cdot m \bmod p]$; note that $[(m-m')^{-1} \bmod p]$ exists because $m \neq m'$ and so $m - m' \neq 0 \bmod p$. Restated, this means that for any m, m', t, t' as above there is a *unique* key (a, b) with $h_{a,b}(m) = t$ and $h_{a,b}(m') = t'$. Since there are $|\mathcal{T}|^2$ keys, we conclude that the probability (over choice of the key) that $h_{a,b}(m) = t$ and $h_{a,b}(m') = t'$ is exactly $1/|\mathcal{K}| = 1/|\mathcal{T}|^2$ as required. ■

Parameters of Construction 4.24. We briefly discuss the parameters of Construction 4.24 when instantiated with the strongly universal function described above, ignoring the fact that p is not a power of 2. The construction is a $1/|\mathcal{T}|$-secure MAC with tags of length $\log |\mathcal{T}|$; the tag length is optimal for the level of security achieved.

Let $\mathcal{M}$ be some fixed message space for which we want to construct a one-time secure MAC. The construction above gives a $1/|\mathcal{M}|$-secure MAC with keys that are twice the length of the messages. The reader may notice two problems here, at opposite ends of the spectrum: First, if $|\mathcal{M}|$ is small then a $1/|\mathcal{M}|$ probability of forgery may be unacceptably large. On the flip side, if $|\mathcal{M}|$ is large then a $1/|\mathcal{M}|$ probability of forgery may be overkill; one might be willing to accept a (somewhat) larger probability of forgery if that level of security can be achieved with shorter keys. The first problem (when $|\mathcal{M}|$ is small) is easy to deal with by simply embedding $\mathcal{M}$ into a larger message space $\mathcal{M}'$ by, for example, padding messages with 0s. The second problem can be addressed as well; see the references at the end of this chapter.

4.6.2 Limitations on Information-Theoretic MACs

In this section we explore limitations on information-theoretic message authentication. We show that any 2^{-n}-secure MAC must have keys of length at least $2n$. An extension of the proof shows that any ℓ-time 2^{-n}-secure MAC (where security is defined via a natural modification of Definition 4.23) requires keys of length at least $(\ell + 1) \cdot n$. A corollary is that *no* MAC with bounded-length keys can provide information-theoretic security when authenticating an unbounded number of messages.

In the following, we assume the message space contains at least two messages; if not, there is no point in communicating, let alone authenticating.

THEOREM 4.27 *Let* $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ *be a* 2^{-n}*-secure MAC where all keys output by* Gen *are the same length. Then the keys output by* Gen *must have length at least* $2n$*.*

PROOF Fix two distinct messages m_0, m_1 in the message space. The intuition for the proof is that there must be at least 2^n possibilities for the tag

of m_0 (or else the adversary could guess it with probability better than 2^{-n}); furthermore, even conditioned on the value of the tag for m_0, there must be 2^n possibilities for the tag of m_1 (or else the adversary could forge a tag on m_1 with probability better than 2^{-n}). Since each key defines tags for m_0 and m_1, this means there must be at least $2^n \times 2^n$ keys. We make this formal below.

Let $\mathcal{K}$ denote the key space (i.e., the set of all possible keys that can be output by Gen). For any possible tag t_0, let $\mathcal{K}(t_0)$ denote the set of keys for which t_0 is a valid tag on m_0; i.e.,

$$\mathcal{K}(t_0) \stackrel{\text{def}}{=} \{k \mid \mathsf{Vrfy}_k(m_0, t_0) = 1\}.$$

For any t_0 we must have $|\mathcal{K}(t_0)| \leq 2^{-n} \cdot |\mathcal{K}|$. Otherwise the adversary could simply output (m_0, t_0) as its forgery; this would be a valid forgery with probability at least $|\mathcal{K}(t_0)|/|\mathcal{K}| > 2^{-n}$, contradicting the claimed security.

Consider now the adversary $\mathcal{A}$ who requests a tag on the message m_0, receives in return a tag t_0, chooses a uniform key $k \in \mathcal{K}(t_0)$, and outputs $(m_1, \mathsf{Mac}_k(m_1))$ as its forgery. The probability that $\mathcal{A}$ outputs a valid forgery is at least

$$\begin{aligned}\sum_{t_0} \Pr[\mathsf{Mac}_k(m_0) = t_0] \cdot \frac{1}{|\mathcal{K}(m_0, t_0)|} &\geq \sum_{t_0} \Pr[\mathsf{Mac}_k(m_0) = t_0] \cdot \frac{2^n}{|\mathcal{K}|} \\ &= \frac{2^n}{|\mathcal{K}|}.\end{aligned}$$

By the claimed security of the scheme, the probability that the adversary can output a valid forgery is at most 2^{-n}. Thus, we must have $|\mathcal{K}| \geq 2^{2n}$. Since all keys have the same length, each key must have length at least $2n$. ■

References and Additional Reading

The definition of security for message authentication codes was adapted by Bellare et al. [18] from the definition of security for digital signatures given by Goldwasser et al. [81] (see Chapter 12). For more on the definitional variant where verification queries are allowed, see [17].

The paradigm of using pseudorandom functions for message authentication (as in Construction 4.5) was introduced by Goldreich et al. [77]. Construction 4.7 is due to Goldreich [76].

CBC-MAC was standardized in the early 1980s [94, 178] and is still used widely today. Basic CBC-MAC was proven secure (for authenticating fixed-length messages) by Bellare et al. [18]. Bernstein [26, 27] gives a more direct (though perhaps less intuitive) proof, and also discusses some generalized

versions of CBC-MAC. As noted in this chapter, basic CBC-MAC is insecure when used to authenticate messages of different lengths. One way to fix this is to prepend the length to the message. This has the disadvantage of not being able to cope with *streaming* data, where the length of the message is not known in advance. Petrank and Rackoff [137] suggest an alternate, "on-line" approach addressing this issue. Further improvements were given by Black and Rogaway [32] and Iwata and Kurosawa [95]; these led to a new proposed standard called CMAC.

The importance of authenticated encryption was first explicitly highlighted in [100, 19], who propose definitions similar to what we have given here. Bellare and Namprempre [19] analyze the three generic approaches discussed here, although the idea of using encrypt-then-authenticate for achieving CCA-security goes back at least to the work of Dolev et al. [61]. Krawczyk [108] examines other methods for achieving secrecy and authentication, and also analyzes the authenticate-then-encrypt approach used by SSL. Degabriele and Paterson [54] show an attack on IPsec when configured to authenticate-then-encrypt (the default for authenticated encryption is actually encrypt-then-authenticate; however it is possible to achieve authenticate-then-encrypt in some configurations). Several nongeneric schemes for authenticated encryption have also been proposed; see [110] for a detailed comparison.

Information-theoretic MACs were first studied by Gilbert et al. [73]. Wegman and Carter [177] introduced the notion of strongly universal functions, and noted their application to one-time message authentication. They also show how to reduce the key length for this task by using an *almost* strongly universal function. Specifically, the construction we give here achieves 2^{-n}-security for messages of length n with keys of length $O(n)$; Wegman and Carter show how to construct a 2^{-n}-secure MAC for messages of length ℓ with keys of (essentially) length $\mathcal{O}(n \cdot \log \ell)$. The simple construction of a strongly universal function that we give here is (with minor differences) due to Carter and Wegman [42]. The reader interested in learning more about information-theoretic MACs is referred to the paper by Stinson [166], the survey by Simmons [162], or the first edition of Stinson's textbook [167, Chapter 10].

Exercises

4.1 Say $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ is a secure MAC, and for $k \in \{0,1\}^n$ the tag-generation algorithm Mac_k always outputs tags of length $t(n)$. Prove that t must be super-logarithmic or, equivalently, that if $t(n) = \mathcal{O}(\log n)$ then Π cannot be a secure MAC.

Hint: Consider the probability of randomly guessing a valid tag.

4.2 Consider an extension of the definition of secure message authentication where the adversary is provided with both a Mac and a Vrfy oracle.

(a) Provide a formal definition of security for this case.

(b) Assume Π is a deterministic MAC using canonical verification that satisfies Definition 4.2. Prove that Π also satisfies your definition from part (a).

4.3 Assume secure MACs exist. Give a construction of a MAC that is secure with respect to Definition 4.2 but that is not secure when the adversary is additionally given access to a Vrfy oracle (cf. the previous exercise).

4.4 Prove Proposition 4.4.

4.5 Assume secure MACs exist. Prove that there exists a MAC that is secure (by Definition 4.2) but is *not* strongly secure (by Definition 4.3).

4.6 Consider the following MAC for messages of length $\ell(n) = 2n - 2$ using a pseudorandom function F: On input a message $m_0 \| m_1$ (with $|m_0| = |m_1| = n - 1$) and key $k \in \{0,1\}^n$, algorithm Mac outputs $t = F_k(0\|m_0) \,\|\, F_k(1\|m_1)$. Algorithm Vrfy is defined in the natural way. Is (Gen, Mac, Vrfy) secure? Prove your answer.

4.7 Let F be a pseudorandom function. Show that each of the following MACs is insecure, even if used to authenticate fixed-length messages. (In each case Gen outputs a uniform $k \in \{0,1\}^n$. Let $\langle i \rangle$ denote an $n/2$-bit encoding of the integer i.)

(a) To authenticate a message $m = m_1, \ldots, m_\ell$, where $m_i \in \{0,1\}^n$, compute $t := F_k(m_1) \oplus \cdots \oplus F_k(m_\ell)$.

(b) To authenticate a message $m = m_1, \ldots, m_\ell$, where $m_i \in \{0,1\}^{n/2}$, compute $t := F_k(\langle 1 \rangle \| m_1) \oplus \cdots \oplus F_k(\langle \ell \rangle \| m_\ell)$.

(c) To authenticate a message $m = m_1, \ldots, m_\ell$, where $m_i \in \{0,1\}^{n/2}$, choose uniform $r \leftarrow \{0,1\}^n$, compute

$$t := F_k(r) \oplus F_k(\langle 1 \rangle \| m_1) \oplus \cdots \oplus F_k(\langle \ell \rangle \| m_\ell),$$

and let the tag be $\langle r, t \rangle$.

4.8 Let F be a pseudorandom function. Show that the following MAC for messages of length $2n$ is insecure: Gen outputs a uniform $k \in \{0,1\}^n$. To authenticate a message $m_1 \| m_2$ with $|m_1| = |m_2| = n$, compute the tag $F_k(m_1) \,\|\, F_k(F_k(m_2))$.

4.9 Given any *deterministic* MAC (Mac, Vrfy), we may view Mac as a keyed function. In both Constructions 4.5 and 4.11, Mac is a pseudorandom function. Give a construction of a secure, deterministic MAC in which Mac is *not* a pseudorandom function.

4.10 Is Construction 4.5 necessarily secure when instantiated using a weak pseudorandom function (cf. Exercise 3.26)? Explain.

4.11 Prove that Construction 4.7 is secure even when the adversary is additionally given access to a Vrfy oracle (cf. Exercise 4.2).

4.12 Prove that Construction 4.7 is secure if it is changed as follows: 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. (Assume for simplicity that the length of all messages being authenticated is always an integer multiple of $n/2 - 1$.) What is the advantage of this modification?

4.13 We explore what happens when the basic CBC-MAC construction is used with messages of different lengths.

(a) Say the sender and receiver do not agree on the message length in advance (and so $\mathsf{Vrfy}_k(m, t) = 1$ iff $t \stackrel{?}{=} \mathsf{Mac}_k(m)$, regardless of the length of m), but the sender is careful to only authenticate messages of length $2n$. Show that an adversary can forge a valid tag on a message of length $4n$.

(b) Say the receiver only accepts 3-block messages (so $\mathsf{Vrfy}_k(m, t) = 1$ only if m has length $3n$ and $t \stackrel{?}{=} \mathsf{Mac}_k(m)$), but the sender authenticates messages of any length a multiple of n. Show that an adversary can forge a valid tag on a new message.

4.14 Prove that the following modifications of basic CBC-MAC do not yield a secure MAC (even for fixed-length messages):

(a) Mac outputs all blocks $t_1, \ldots, t_\ell$, rather than just t_ℓ. (Verification only checks whether t_ℓ is correct.)

(b) A random initial block is used each time a message is authenticated. That is, choose uniform $t_0 \in \{0,1\}^n$, run basic CBC-MAC over the "message" $t_0, m_1, \ldots, m_\ell$, and output the tag $\langle t_0, t_\ell \rangle$. Verification is done in the natural way.

4.15 Show that appending the message length to the *end* of the message before applying basic CBC-MAC does not result in a secure MAC for arbitrary-length messages.

4.16 Show that the encoding for arbitrary-length messages described in Section 4.4.2 is prefix-free.

4.17 Consider the following encoding that handles messages whose length is less than $n \cdot 2^n$: We encode a string $m \in \{0,1\}^*$ by first appending as many 0s as needed to make the length of the resulting string $\hat{m}$ a nonzero multiple of n. Then we prepend the number of *blocks* in $\hat{m}$ (equivalently, prepend the integer $|\hat{m}|/n$), encoded as an n-bit string. Show that this encoding is *not* prefix-free.

4.18 Prove that the following modification of basic CBC-MAC gives a secure MAC for arbitrary-length messages (for simplicity, assume all messages have length a multiple of the block length). $\mathsf{Mac}_k(m)$ first computes $k_\ell = F_k(\ell)$, where ℓ is the length of m. The tag is then computed using basic CBC-MAC with key k_ℓ. Verification is done in the natural way.

4.19 Let F be a keyed function that is a secure (deterministic) MAC for messages of length n. (Note that F need not be a pseudorandom permutation.) Show that basic CBC-MAC is not necessarily a secure MAC (even for fixed-length messages) when instantiated with F.

4.20 Show that Construction 4.7 is strongly secure.

4.21 Show that Construction 4.18 might not be CCA-secure if it is instantiated with a secure MAC that is *not* strongly secure.

4.22 Prove that Construction 4.18 is unforgeable when instantiated with any encryption scheme (even if not CPA-secure) and any secure MAC (even if the MAC is not strongly secure).

4.23 Consider a strengthened version of unforgeability (Definition 4.16) where $\mathcal{A}$ is additionally given access to a decryption oracle.

(a) Write a formal definition for this version of unforgeability.

(b) Prove that Construction 4.18 satisfies this stronger definition if Π_M is a strongly secure MAC.

(c) Show by counterexample that Construction 4.18 need not satisfy this stronger definition if Π_M is a secure MAC that is not strongly secure. (Compare to the previous exercise.)

4.24 Prove that the authenticate-then-encrypt approach, instantiated with any CPA-secure encryption scheme and any secure MAC, yields a CPA-secure encryption scheme that is unforgeable.

4.25 Let F be a strong pseudorandom permutation, and define the following fixed-length encryption scheme: On input a message $m \in \{0,1\}^{n/2}$ and key $k \in \{0,1\}^n$, algorithm Enc chooses a uniform $r \in \{0,1\}^{n/2}$ and computes $c := F_k(m\|r)$. (See Exercise 3.18.) Prove that this scheme is CCA-secure, but is not an authenticated encryption scheme.

4.26 Show a CPA-secure private-key encryption scheme that is unforgeable but is not CCA-secure.

4.27 Fix $\ell > 0$ and a prime p. Let $\mathcal{K} = \mathbb{Z}_p^{\ell+1}$, $\mathcal{M} = \mathbb{Z}_p^{\ell}$, and $\mathcal{T} = \mathbb{Z}_p$. Define $h : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ as

$$h_{k_0,k_1,\ldots,k_\ell}(m_1,\ldots,m_\ell) = \left[k_0 + \textstyle\sum_i k_i m_i \bmod p\right].$$

Prove that h is strongly universal.

4.28 Fix $\ell, n > 0$. Let $\mathcal{K} = \{0,1\}^{\ell \times n} \times \{0,1\}^{\ell}$ (interpreted as a boolean $\ell \times n$ matrix and an ℓ-dimensional vector), let $\mathcal{M} = \{0,1\}^n$, and let $\mathcal{T} = \{0,1\}^{\ell}$. Define $h : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ as $h_{K,v}(m) = K \cdot m \oplus v$, where all operations are performed modulo 2. Prove that h is strongly universal.

4.29 A *Toeplitz matrix* K is a matrix in which $K_{i,j} = K_{i-1,j-1}$ when $i, j > 1$; i.e., the values along any diagonal are equal. So an $\ell \times n$ Toeplitz matrix (for $\ell > n$) has the form

$$\begin{bmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \cdots & K_{1,n} \\ K_{2,1} & K_{1,1} & K_{1,2} & \cdots & K_{1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ K_{\ell,1} & K_{\ell-1,1} & K_{\ell-2,1} & \cdots & K_{\ell-n+1,1} \end{bmatrix}.$$

Let $\mathcal{K} = T^{\ell \times n} \times \{0,1\}^{\ell}$ (where $T^{\ell \times n}$ denotes the set of $\ell \times n$ Toeplitz matrices), let $\mathcal{M} = \{0,1\}^n$, and let $\mathcal{T} = \{0,1\}^{\ell}$. Define $h : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ as $h_{K,v}(m) = K \cdot m \oplus v$, where all operations are performed modulo 2. Prove that h is strongly universal. What is the advantage here as compared to the construction in the previous exercise?

4.30 Define an appropriate notion of a *two-time* ε-secure MAC, and give a construction that meets your definition.

4.31 Let $\{h_n : \mathcal{K}_n \times \{0,1\}^{10 \cdot n} \to \{0,1\}^n\}_{n \in \mathbb{N}}$ be such that h_n is strongly universal for all n, and let F be a pseudorandom function. (When $K \in \mathcal{K}_n$ we write $h_K(\cdot)$ instead of $h_{n,K}(\cdot)$.) Consider the following MAC: $\mathsf{Gen}(1^n)$ chooses uniform $K \in \mathcal{K}_n$ and $k \in \{0,1\}^n$, and outputs (K, k). To authenticate a message $m \in \{0,1\}^{10 \cdot n}$, choose uniform $r \in \{0,1\}^n$ and output $\langle r,\ h_K(m) \oplus F_k(r) \rangle$. Verification is done in the natural way. Prove that this gives a (computationally) secure MAC for messages of length $10n$.

Chapter 5

Hash Functions and Applications

In this chapter we introduce *cryptographic hash functions* and explore a few of their applications. At the most basic level, a hash function provides a way to map a long input string to a shorter output string sometimes called a *digest*. The primary requirement is to avoid *collisions*, or two inputs that map to the same digest. Collision-resistant hash functions have numerous uses. One example that we will see here is another approach—standardized as HMAC—for achieving domain extension for message authentication codes.

Beyond that, hash functions have become ubiquitous in cryptography, and they are often used in scenarios that require properties much stronger than collision resistance. It has become common to model cryptographic hash functions as being "completely unpredictable" (a.k.a., *random oracles*), and we discuss this framework—and the controversy that surrounds it—in detail later in the chapter. We touch on only a few applications of the random-oracle model here, but will encounter it again when we turn to the setting of public-key cryptography.

Hash functions are intriguing in that they can be viewed as lying between the worlds of private- and public-key cryptography. On the one hand, as we will see in Chapter 6, they are (in practice) constructed using symmetric-key techniques, and many of the canonical applications of hash functions are in the symmetric-key setting. From a theoretical point of view, however, the existence of collision-resistant hash functions appears to represent a qualitatively stronger assumption than the existence of pseudorandom functions (yet a weaker assumption than the existence of public-key encryption).

5.1 Definitions

Hash functions are simply functions that take inputs of some length and *compress* them into short, fixed-length outputs. The classic use of hash functions is in data structures, where they can be used to build hash tables that enable $\mathcal{O}(1)$ lookup time when storing a set of elements. Specifically, if the range of the hash function H is of size N, then element x is stored in row $H(x)$ of a table of size N. To retrieve x, it suffices to compute $H(x)$ and probe

that row of the table for the elements stored there. A "good" hash function for this purpose is one that yields few *collisions*, where a collision is a pair of distinct items x and x' for which $H(x) = H(x')$; in this case we also say that x and x' *collide*. (When a collision occurs, two elements end up being stored in the same cell, increasing the lookup time.)

Collision-resistant hash functions are similar in spirit. Again, the goal is to avoid collisions. However, there are fundamental differences. For one, the desire to minimize collisions in the setting of data structures becomes a *requirement* to avoid collisions in the setting of cryptography. Furthermore, in the context of data structures we can assume that the set of data elements is chosen independently of the hash function and without any intention to cause collisions. In the context of cryptography, in contrast, we are faced with an adversary who may select elements with the explicit goal of causing collisions. This means that collision-resistant hash functions are much harder to design.

5.1.1 Collision Resistance

Informally, a function H is *collision resistant* if it is infeasible for any probabilistic polynomial-time algorithm to find a collision in H. We will only be interested in hash functions whose domain is larger than their range. In this case collisions must *exist*, but such collisions should be hard to find.

Formally, we consider *keyed* hash functions. That is, H is a two-input function that takes as input a key s and a string x, and outputs a string $H^s(x) \stackrel{\text{def}}{=} H(s, x)$. The requirement is that it must be hard to find a collision in H^s for a randomly generated key s. There are at least two differences between keys in this context and keys as we have used them until now. First, not all strings necessarily correspond to valid keys (i.e., H^s may not be defined for certain s), and therefore the key s will typically be generated by an algorithm Gen rather than being chosen uniformly. Second, and perhaps more importantly, this key s is (generally) not kept secret, and collision resistance is required even when the adversary is given s. In order to emphasize this, we superscript the key and write H^s rather than H_s.

DEFINITION 5.1 *A* hash function *(with output length ℓ) is a pair of probabilistic polynomial-time algorithms (Gen, H) satisfying the following:*

- Gen *is a probabilistic algorithm which takes as input a security parameter 1^n and outputs a key s. We assume that 1^n is implicit in s.*
- *H takes as input a key s and a string $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{\ell(n)}$ (where n is the value of the security parameter implicit in s).*

If H^s is defined only for inputs $x \in \{0,1\}^{\ell'(n)}$ and $\ell'(n) > \ell(n)$, then we say that (Gen, H) is a fixed-length hash function for inputs of length ℓ'. *In this case, we also call H a* compression function.

In the fixed-length case we require that ℓ' be greater than ℓ. This ensures that the function *compresses* its input. In the general case the function takes as input strings of arbitrary length. Thus, it also compresses (albeit only strings of length greater than $\ell(n)$). Note that without compression, collision resistance is trivial (since one can just take the identity function $H^s(x) = x$).

We now proceed to define security. As usual, we first define an experiment for a hash function $\Pi = (\mathsf{Gen}, H)$, an adversary $\mathcal{A}$, and a security parameter n:

The collision-finding experiment $\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n)$**:**

1. *A key s is generated by running* $\mathsf{Gen}(1^n)$.
2. *The adversary $\mathcal{A}$ is given s and outputs x, x'. (If Π is a fixed-length hash function for inputs of length $\ell'(n)$, then we require $x, x' \in \{0,1\}^{\ell'(n)}$.)*
3. *The output of the experiment is defined to be* 1 *if and only if $x \neq x'$ and $H^s(x) = H^s(x')$. In such a case we say that $\mathcal{A}$ has found a collision.*

The definition of collision resistance states that no efficient adversary can find a collision in the above experiment except with negligible probability.

DEFINITION 5.2 *A hash function $\Pi = (\mathsf{Gen}, H)$ is* collision resistant *if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there is a negligible function* negl *such that*

$$\Pr\left[\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n) = 1\right] \leq \mathsf{negl}(n).$$

For simplicity, we sometimes refer to H or H^s as a "collision-resistant hash function," even though technically we should only say that (Gen, H) is. This should not cause any confusion.

Cryptographic hash functions are designed with the explicit goal of being collision resistant (among other things). We will discuss some common real-world hash functions in Chapter 6. In Section 8.4.2 we will see how it is possible to construct hash functions with proven collision resistance based on an assumption about the hardness of a certain number-theoretic problem.

Unkeyed hash functions. Cryptographic hash functions used in practice generally have a fixed output length (just as block ciphers have a fixed key length) and are usually *unkeyed*, meaning that the hash function is just a fixed function $H : \{0,1\}^* \to \{0,1\}^\ell$. This is problematic from a theoretical standpoint since for any such function there is always a constant-time algorithm that outputs a collision in H: the algorithm simply outputs a colliding pair (x, x') hardcoded into the algorithm itself. Using keyed hash functions solves this technical issue since it is impossible to hardcode a colliding pair for every possible key using a reasonable amount of space (and in an asymptotic setting, it would be impossible to hardcode a colliding pair for every value of the security parameter).

Notwithstanding the above, the (unkeyed) cryptographic hash functions used in the real world are collision resistant for all practical purposes since colliding pairs are unknown (and computationally difficult to find) even though they must exist. Proofs of security for some construction based on collision resistance of a hash function are meaningful even when an unkeyed hash function H is used, as long as the proof shows that any efficient adversary "breaking" the primitive can be used to efficiently find a collision in H. (All the proofs in this book satisfy this condition.) In this case, the interpretation of the proof of security is that if an adversary can break the scheme in practice, then it can be used to find a collision in practice, something that we believe is hard to do.

5.1.2 Weaker Notions of Security

In some applications it suffices to rely on security requirements weaker than collision resistance. These include:

- *Second-preimage or target-collision resistance:* Informally, a hash function is second preimage resistant if given s and a uniform x it is infeasible for a PPT adversary to find $x' \neq x$ such that $H^s(x') = H^s(x)$.
- *Preimage resistance:* Informally, a hash function is preimage resistant if given s and a uniform y it is infeasible for a PPT adversary to find a value x such that $H^s(x) = y$. (Looking ahead to Chapter 7, this essentially means that H^s is *one-way*.)

Any hash function that is collision resistant is also second preimage resistant. This holds since if, given a uniform x, an adversary can find $x' \neq x$ for which $H^s(x') = H^s(x)$, then it can clearly find a colliding pair x and x'. Likewise, any hash function that is second preimage resistant is also preimage resistant. This is due to the fact that if it were possible, given y, to find an x such that $H^s(x) = y$, then one could also take a given input x', compute $y := H^s(x')$, and then obtain an x with $H^s(x) = y$. With high probability $x' \neq x$ (relying on the fact that H compresses, and so multiple inputs map to the same output), in which case a second preimage has been found.

We do not formally define the above notions or prove the above implications, since they are not used in the rest of the book. You are asked to formalize the above in Exercise 5.1.

5.2 Domain Extension: The Merkle-Damgård Transform

Hash functions are often constructed by first designing a collision-resistant compression function handling fixed-length inputs, and then using *domain*

extension to handle arbitrary-length inputs. In this section, we show one solution to the problem of domain extension. We return to the question of designing collision-resistant compression functions in Section 6.3.

The *Merkle–Damgård transform* is a common approach for extending a compression function to a full-fledged hash function, while maintaining the collision-resistance property of the former. It is used extensively in practice for hash functions including MD5 and the SHA family (see Section 6.3). The existence of this transform means that when designing collision-resistant hash functions, we can restrict our attention to the fixed-length case. This, in turn, makes the job of designing collision-resistant hash functions much easier. The Merkle–Damgård transform is also interesting from a theoretical point of view since it implies that compressing by a single bit is as easy (or as hard) as compressing by an arbitrary amount.

For concreteness, assume the compression function (Gen, h) compresses its input by half; say its input length is $2n$ and its output length is n. (The construction works regardless of the input/output lengths, as long as h compresses.) We construct a collision-resistant hash function (Gen, H) that maps inputs of *arbitrary* length to outputs of length n. (Gen remains unchanged.) The Merkle–Damgård transform is defined in Construction 5.3 and depicted in Figure 5.1. The value z_0 used in step 2 of the construction, called the *initialization vector* or IV, is arbitrary and can be replaced by any constant.

CONSTRUCTION 5.3

Let (Gen, h) be a fixed-length hash function for inputs of length $2n$ and with output length n. Construct hash function (Gen, H) as follows:

- Gen: remains unchanged.
- H: on input a key s and a string $x \in \{0,1\}^*$ of length $L < 2^n$, do the following:
  1. Set $B := \lceil \frac{L}{n} \rceil$ (i.e., the number of blocks in x). Pad x with zeros so its length is a multiple of n. Parse the padded result as the sequence of n-bit blocks $x_1, \ldots, x_B$. Set $x_{B+1} := L$, where L is encoded as an n-bit string.
  2. Set $z_0 := 0^n$. (This is also called the IV.)
  3. For $i = 1, \ldots, B+1$, compute $z_i := h^s(z_{i-1} \| x_i)$.
  4. Output z_{B+1}.

The Merkle–Damgård transform.

THEOREM 5.4 *If* (Gen, h) *is collision resistant, then so is* (Gen, H).

PROOF We show that for any s, a collision in H^s yields a collision in h^s. Let x and x' be two different strings of length L and L', respectively, such that $H^s(x) = H^s(x')$. Let $x_1, \ldots, x_B$ be the B blocks of the padded x, and

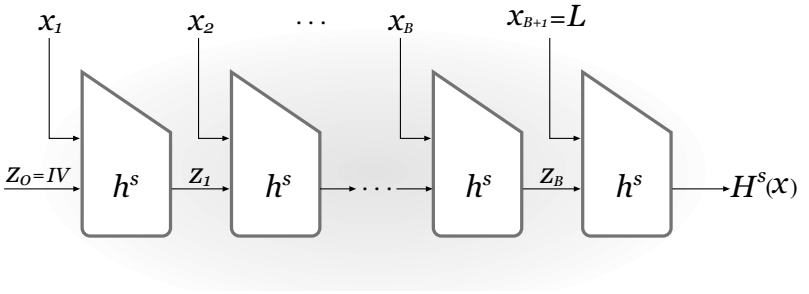


FIGURE 5.1: The Merkle–Damgård transform.

let $x'_1, \ldots, x'_{B'}$ be the B' blocks of the padded x'. Recall that $x_{B+1} = L$ and $x'_{B'+1} = L'$. There are two cases to consider:

1. *Case 1:* $L \neq L'$. In this case, the last step of the computation of $H^s(x)$ is $z_{B+1} := h^s(z_B \| L)$, and the last step of the computation of $H^s(x')$ is $z'_{B'+1} := h^s(z'_{B'} \| L')$. Since $H^s(x) = H^s(x')$ it follows that $h^s(z_B \| L) = h^s(z'_{B'} \| L')$. However, $L \neq L'$ and so $z_B \| L$ and $z'_{B'} \| L'$ are two different strings that collide under h^s.

2. *Case 2:* $L = L'$. This means that $B = B'$. Let $z_0, \ldots, z_{B+1}$ be the values defined during the computation of $H^s(x)$, let $I_i \stackrel{\text{def}}{=} z_{i-1} \| x_i$ denote the ith input to h^s, and set $I_{B+2} \stackrel{\text{def}}{=} z_{B+1}$. Define $I'_1, \ldots, I'_{B+2}$ analogously with respect to x'. Let N be the *largest* index for which $I_N \neq I'_N$. Since $|x| = |x'|$ but $x \neq x'$, there is an i with $x_i \neq x'_i$ and so such an N certainly exists. Because

$$I_{B+2} = z_{B+1} = H^s(x) = H^s(x') = z'_{B+1} = I'_{B+2},$$

   we have $N \leq B + 1$. By maximality of N, we have $I_{N+1} = I'_{N+1}$ and in particular $z_N = z'_N$. But this means that I_N, I'_N are a collision in h^s.

We leave it as an exercise to turn the above into a formal reduction. ■

5.3 Message Authentication Using Hash Functions

In the previous chapter, we presented two constructions of message authentication codes for arbitrary-length messages. The first approach was generic, but inefficient. The second, CBC-MAC, was based on pseudorandom functions. Here we will see another approach, which we call "hash-and-MAC," that relies on collision-resistant hashing along with any message authentication code. We then discuss a standardized and widely used construction called HMAC that can be viewed as a specific instantiation of this approach.

5.3.1 Hash-and-MAC

The idea behind the hash-and-MAC approach is simple. First, an arbitrarily long message m is hashed down to a fixed-length string $H^s(m)$ using a collision-resistant hash function. Then, a (fixed-length) MAC is applied to the result. See Construction 5.5 for a formal description.

CONSTRUCTION 5.5

Let $\Pi = (\mathsf{Mac}, \mathsf{Vrfy})$ be a MAC for messages of length $\ell(n)$, and let $\Pi_H = (\mathsf{Gen}_H, H)$ be a hash function with output length $\ell(n)$. Construct a MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ for arbitrary-length messages as follows:

- Gen': on input 1^n, choose uniform $k \in \{0,1\}^n$ and run $\mathsf{Gen}_H(1^n)$ to obtain s; the key is $k' := \langle k, s\rangle$.
- Mac': on input a key $\langle k, s\rangle$ and a message $m \in \{0,1\}^*$, output $t \leftarrow \mathsf{Mac}_k(H^s(m))$.
- Vrfy': on input a key $\langle k, s\rangle$, a message $m \in \{0,1\}^*$, and a MAC tag t, output 1 if and only if $\mathsf{Vrfy}_k(H^s(m), t) \stackrel{?}{=} 1$.

The hash-and-MAC paradigm.

Construction 5.5 is secure if Π is a secure MAC for fixed-length messages and (Gen, H) is collision resistant. Intuitively, since the hash function is collision resistant, authenticating $H^s(m)$ is as good as authenticating m itself: if the sender can ensure that the receiver obtains the correct value $H^s(m)$, collision resistance guarantees that the attacker cannot find a different message m' that hashes to the same value. A bit more formally, say a sender uses Construction 5.5 to authenticate some set of messages $\mathcal{Q}$, and an attacker $\mathcal{A}$ is then able to forge a valid tag on a new message $m^* \notin \mathcal{Q}$. There are two possible cases:

Case 1: *there is a message $m \in \mathcal{Q}$ such that $H^s(m^*) = H^s(m)$.* Then $\mathcal{A}$ has found a collision in H^s, contradicting the collision resistance of (Gen, H).

Case 2: *for every message $m \in \mathcal{Q}$ it holds that $H^s(m^*) \neq H^s(m)$.* Let $H^s(\mathcal{Q}) \stackrel{\text{def}}{=} \{H^s(m) \mid m \in \mathcal{Q}\}$. Then $H^s(m^*) \notin H^s(\mathcal{Q})$. In this case, $\mathcal{A}$ has forged a valid tag on the "new message" $H^s(m^*)$ with respect to the *fixed-length* message authentication code Π. This contradicts the assumption that Π is a secure MAC.

We now turn the above into a formal proof.

THEOREM 5.6 *If Π is a secure MAC for messages of length ℓ and Π_H is collision resistant, then Construction 5.5 is a secure MAC (for arbitrary-length messages).*

PROOF Let Π' denote Construction 5.5, and let $\mathcal{A}'$ be a PPT adversary attacking Π'. In an execution of experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$, let $k' = \langle k, s\rangle$ denote the MAC key, let $\mathcal{Q}$ denote the set of messages whose tags were requested by $\mathcal{A}'$, and let (m^*, t) be the final output of $\mathcal{A}'$. We assume without loss of generality that $m^* \notin \mathcal{Q}$. Define coll to be the event that, in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$, there is an $m \in \mathcal{Q}$ for which $H^s(m^*) = H^s(m)$. We have

$$\begin{aligned}
&\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1] \\
&= \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \mathsf{coll}] + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\mathsf{coll}}] \\
&\leq \Pr[\mathsf{coll}] + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\mathsf{coll}}]. \qquad (5.1)
\end{aligned}$$

We show that both terms in Equation (5.1) are negligible, thus completing the proof. Intuitively, the first term is negligible by collision resistance of Π_H, and the second term is negligible by security of Π.

Consider the following algorithm $\mathcal{C}$ for finding a collision in Π_H:

Algorithm $\mathcal{C}$:
The algorithm is given s as input (with n implicit).

- Choose uniform $k \in \{0,1\}^n$.
- Run $\mathcal{A}'(1^n)$. When $\mathcal{A}'$ requests a tag on the ith message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \mathsf{Mac}_k(H^s(m_i))$ and give t_i to $\mathcal{A}'$.
- When $\mathcal{A}'$ outputs (m^*, t), then if there exists an i for which $H^s(m^*) = H^s(m_i)$, output (m^*, m_i).

It is clear that $\mathcal{C}$ runs in polynomial time. Let us analyze its behavior. When the input to $\mathcal{C}$ is generated by running $\mathsf{Gen}_H(1^n)$ to obtain s, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{C}$ is distributed identically to the view of $\mathcal{A}'$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. In particular, the tags given to $\mathcal{A}'$ by $\mathcal{C}$ have the same distribution as the tags that $\mathcal{A}'$ receives in $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. Since $\mathcal{C}$ outputs a collision exactly when coll occurs, we have

$$\Pr[\mathsf{Hash\text{-}coll}_{\mathcal{C},\Pi_H}(n) = 1] = \Pr[\mathsf{coll}].$$

Because Π_H is collision resistant, we conclude that $\Pr[\mathsf{coll}]$ is negligible.

We now proceed to prove that the second term in Equation (5.1) is negligible. Consider the following adversary $\mathcal{A}$ attacking Π in $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$:

Adversary $\mathcal{A}$:
The adversary is given access to a MAC oracle $\mathsf{Mac}_k(\cdot)$.

- Compute $\mathsf{Gen}_H(1^n)$ to obtain s.
- Run $\mathcal{A}'(1^n)$. When $\mathcal{A}'$ requests a tag on the ith message $m_i \in \{0,1\}^*$, then: (1) compute $\hat{m}_i := H^s(m_i)$; (2) obtain a tag t_i on $\hat{m}_i$ from the MAC oracle; and (3) give t_i to $\mathcal{A}'$.
- When $\mathcal{A}'$ outputs (m^*, t), then output $(H^s(m^*), t)$.

Clearly $\mathcal{A}$ runs in polynomial time. Consider experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$. In that experiment, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{A}$ is distributed identically to its view in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. Furthermore, whenever both $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1$ and coll do not occur, $\mathcal{A}$ outputs a valid forgery. (In that case t is a valid tag on $H^s(m^*)$ in scheme Π with respect to k. The fact that coll did not occur means that $H^s(m^*)$ was never asked by $\mathcal{A}$ to its own MAC oracle and so this is indeed a forgery.) Therefore,

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) \wedge \overline{\mathsf{coll}}],$$

and security of Π implies that the former probability is negligible. This concludes the proof of the theorem. ∎

5.3.2 HMAC

All the constructions of message authentication codes we have seen so far are ultimately based on some block cipher. Is it possible to construct a secure MAC (for arbitrary-length messages) based directly on a hash function? A first thought might be to define $\mathsf{Mac}_k(m) = H(k\|m)$; we might expect that if H is a "good" hash function then it should be difficult for an attacker to predict the value of $H(k\|m')$ given the value of $H(k\|m)$, for any $m' \neq m$, assuming k is chosen at random (and unknown to the attacker). Unfortunately, if H is constructed using the Merkle–Damgård transform—as most real-world hash functions are—then a MAC designed in this way is completely insecure, as you are asked to show in Exercise 5.10.

Instead, we can try using *two* layers of hashing. See Construction 5.7 for a standardized scheme called *HMAC* based on this idea.

CONSTRUCTION 5.7

Let (Gen_H, H) be a hash function constructed by applying the Merkle–Damgård transform to a compression function (Gen_H, h) taking inputs of length $n + n'$. (See text.) Let opad and ipad be fixed constants of length n'. Define a MAC as follows:

- Gen: on input 1^n, run $\mathsf{Gen}_H(1^n)$ to obtain a key s. Also choose uniform $k \in \{0,1\}^{n'}$. Output the key $\langle s, k\rangle$.
- Mac: on input a key $\langle s, k\rangle$ and a message $m \in \{0,1\}^*$, output

$$t := H^s\Big((k \oplus \mathsf{opad}) \,\|\, H^s\big((k \oplus \mathsf{ipad}) \,\|\, m\big)\Big).$$

- Vrfy: on input a key $\langle s, k\rangle$, a message $m \in \{0,1\}^*$, and a tag t, output 1 if and only if $t \stackrel{?}{=} H^s\big((k \oplus \mathsf{opad}) \,\|\, H^s\big((k \oplus \mathsf{ipad}) \,\|\, m\big)\big)$.

HMAC.

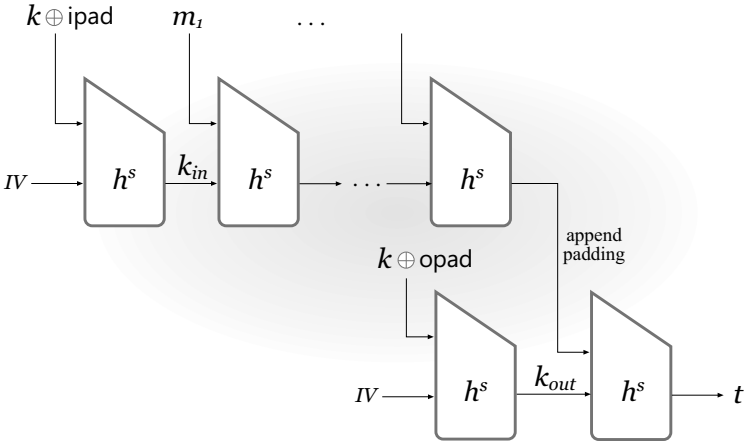


FIGURE 5.2: HMAC, pictorially.

Why should we have any confidence that HMAC is secure? One reason is that we can view HMAC as a specific instantiation of the hash-and-MAC paradigm from the previous section. To see this, we will look "under the hood" at what happens when a message is authenticated; see Figure 5.2. We must also specify parameters more carefully and go into a bit more detail regarding the way the Merkle–Damgård transform is implemented in practice.

Say (Gen_H, H) is constructed based on a compression function (Gen_H, h) in which h maps inputs of length $n + n'$ to outputs of length n (where, formally, n' is a function of n). When we described the Merkle–Damgård transform in Section 5.2, we assumed $n' = n$, but that need not always be the case. We also said that the length of the message being hashed was encoded as an extra message block that is appended to the message. In practice, the length is instead encoded in a *portion* of a block using $\ell < n'$ bits. That is, computation of $H^s(x)$ begins by padding x with zeros to a string of length exactly ℓ less than a multiple of n'; it then appends the length $L = |x|$, encoded using exactly ℓ bits. The hash of the resulting sequence of n'-bit blocks $x_1, \ldots$ is then computed as in Construction 5.3. We will assume that $n + \ell \leq n'$. This means, in particular, that if we hash an input x of length $n' + n$ then the padded result (including the length) will be exactly $2n'$ bits long. The proof of Theorem 5.4, showing that (Gen_H, H) is collision resistant if (Gen_H, h) is collision resistant, remains unchanged.

Coming back to HMAC, and looking at Figure 5.2, we can see that the general form of HMAC involves hashing an arbitrary-length message down to a short string $y \stackrel{\text{def}}{=} H^s(\,(k \oplus \mathsf{ipad}) \,\|\, m)$, and then computing the (secretly keyed) function $H^s((k \oplus \mathsf{opad}) \,\|\, y)$ of the result. But we can say more than this. Note first that the "inner" computation

$$\widetilde{H}^s(m) \stackrel{\text{def}}{=} H^s(\,(k \oplus \mathsf{ipad}) \,\|\, m)$$

is collision resistant (assuming h is), for any value of $k \oplus \mathsf{ipad}$. Moreover, the first step in the "outer" computation $H^s((k \oplus \mathsf{opad}) \| y)$ is to compute a value $k_{out} \stackrel{\text{def}}{=} h^s(IV \| (k \oplus \mathsf{opad}))$. Then, we evaluate $h^s(k_{out} \| \hat{y})$ where $\hat{y}$ refers to the padded value of y (i.e., including the length of $(k \oplus \mathsf{opad}) \| y$, which is always $n' + n$ bits, encoded using exactly ℓ bits). Thus, if we treat k_{out} as uniform—we will be more formal about this below—and assume that

$$\widetilde{\mathsf{Mac}}_k(y) \stackrel{\text{def}}{=} h^s(k \| \hat{y}) \tag{5.2}$$

is a secure *fixed-length* MAC, then HMAC can be viewed as an instantiation of the hash-and-MAC approach with

$$\text{HMAC}_{s,k}(m) = \widetilde{\mathsf{Mac}}_{k_{out}}(\tilde{H}^s(m)) \tag{5.3}$$

(where $k_{out} = h^s(IV \| (k \oplus \mathsf{opad}))$). Because of the way the compression function h is typically designed (see Section 6.3.1), the assumption that $\widetilde{\mathsf{Mac}}$ is a secure fixed-length MAC is a reasonable one.

The roles of ipad and opad. Given the above, one might wonder why it is necessary to incorporate k in the "inner" computation $H^s((k \oplus \mathsf{ipad}) \| m)$. (In particular, for the hash-and-MAC approach to be secure we require collision resistance in the first step, which does not require any secret key.) The reason is that this allows security of HMAC to be based on the potentially weaker assumption that (Gen_H, H) is *weakly* collision resistant, where weak collision resistance is defined by the following experiment: a key s is generated using Gen_H and a uniform *secret* $k_{in} \in \{0,1\}^n$ is chosen. Then the adversary is allowed to interact with a "hash oracle" that returns $H^s_{k_{in}}(m)$ in response to the query m, where $H^s_{k_{in}}$ refers to computation of H^s using the Merkle–Damgård transform applied to h^s, but using the secret value k_{in} as the IV. (Refer again to Figure 5.2.) The adversary succeeds if it can output distinct values m, m' such that $H^s_{k_{in}}(m) = H^s_{k_{in}}(m')$, and we say that (Gen_H, H) is *weakly collision resistant* if every PPT $\mathcal{A}$ succeeds in this experiment with only negligible probability. If (Gen_H, H) is collision resistant then it is clearly weakly collision resistant; the latter, however, is a weaker condition that is potentially easier to satisfy. This is a good example of sound security engineering. This defensive design strategy paid off when it was discovered that the hash function MD5 (see Section 6.3.2) was *not* collision resistant. The collision-finding attacks on MD5 did not violate weak collision resistance, and HMAC-MD5 was not broken even though MD5 was. This gave developers time to replace MD5 in HMAC implementations, without immediate fear of attack. (Despite this, HMAC-MD5 should no longer be used now that weaknesses in MD5 are known.)

The above discussion suggests that *independent* keys should be used in the outer and inner computations. For reasons of efficiency, a single key k is used for HMAC, but the key is used in combination with ipad and opad to derive

two other keys. Define

$$G^s(k) \stackrel{\text{def}}{=} h^s\left(IV \| (k \oplus \mathsf{opad})\right) \| h^s\left(IV \| (k \oplus \mathsf{ipad})\right) = k_{out} \| k_{in}. \qquad (5.4)$$

If we assume that G^s is a pseudorandom generator for any s, then k_{out} and k_{in} can be treated as independent and uniform keys when k is uniform. Security of HMAC then reduces to the security of the following construction:

$$\mathsf{Mac}_{s,k_{in},k_{out}}(m) = h^s\left(k_{out} \| H^s_{k_{in}}(m)\right).$$

(Compare to Equation (5.3).) As noted earlier, this construction can be proven secure (using a variant of the proof for the hash-and-MAC approach) if H is weakly collision resistant and the MAC defined in Equation (5.2) is a secure fixed-length MAC.

THEOREM 5.8 *Assume G^s as defined in Equation (5.4) is a pseudorandom generator for any s, the MAC defined in Equation (5.2) is a secure fixed-length MAC for messages of length n, and (Gen_H, H) is weakly collision resistant. Then HMAC is a secure MAC (for arbitrary-length messages).*

HMAC in practice. HMAC is an industry standard and is widely used in practice. It is highly efficient and easy to implement, and is supported by a proof of security based on assumptions that are believed to hold for practical hash functions. The importance of HMAC is partially due to the timeliness of its appearance. Before the introduction of HMAC, many practitioners refused to use CBC-MAC (with the claim that it was "too slow") and instead used heuristic constructions that were insecure. HMAC provided a standardized, secure way of doing message authentication based on hash functions.

5.4 Generic Attacks on Hash Functions

What is the best security we can hope for a hash function H to provide? We explore this question by showing two attacks that are *generic* in the sense that they apply to *arbitrary* hash functions. The existence of these attacks implies lower bounds on the output length of H needed to achieve some desired level of security, and therefore has important practical ramifications.

5.4.1 Birthday Attacks for Finding Collisions

Let $H : \{0,1\}^* \to \{0,1\}^\ell$ be a hash function. (Here and in the rest of the chapter, we drop explicit mention of the hash key s since it is not directly relevant. One can also view s as being generated and fixed before these

algorithms are applied.) There is a trivial collision-finding attack running in time $\mathcal{O}(2^\ell)$: simply evaluate H on $2^\ell + 1$ distinct inputs; by the pigeonhole principle, two of the outputs must be equal. Can we do better?

Generalizing the above algorithm, say we choose q distinct inputs $x_1, \ldots, x_q$, compute $y_i := H(x_i)$, and check whether any of the two y_i values are equal. What is the probability that this algorithm finds a collision? As we have just said, if $q > 2^\ell$ then a collision occurs with probability 1. What is the probability of a collision when q is smaller? It is somewhat difficult to analyze this probability exactly, and so we will instead analyze an idealized case in which H is treated as a random function.[1] That is, for each i we assume that the value $y_i = H(x_i)$ is uniformly distributed in $\{0,1\}^\ell$ and independent of any of the previous output values $\{y_j\}_{j<i}$ (recall we assume all $\{x_i\}$ are distinct). We have thus reduced our problem to the following one: if we choose values $y_1, \ldots, y_q \in \{0,1\}^\ell$ 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* discussed in detail in Appendix A.4. For this reason, the collision-finding algorithm we have described is often called a *birthday attack*. The birthday problem is the following: if q people are in a room, what is the probability that two of them have the same birthday? (Assume birthdays are uniformly and independently distributed among the 365 days of a non-leap year.) This is exactly analogous to our problem: if y_i represents the birthday of person i, then we have $y_1, \ldots, y_q \in \{1, \ldots, 365\}$ chosen uniformly, and matching birthdays correspond to distinct i, j with $y_i = y_j$ (i.e., matching birthdays correspond to collisions).

In Appendix A.4 we show that for $y_1, \ldots, y_q$ chosen uniformly in $\{1, \ldots, N\}$, the probability of a collision is roughly $1/2$ when $q = \Theta(N^{1/2})$. In the case of birthdays, once there are only 23 people the probability that some two of them have the same birthday is greater than $1/2$. In our setting, this means that when the hash function has output length ℓ (and so the range is of size 2^ℓ), then taking $q = \Theta(2^{\ell/2})$ yields a collision with probability roughly $1/2$.

From a concrete-security perspective, the above means that for a hash function to resist collision-finding attacks that run in time T (where we take the time to evaluate H as our unit of time), the output length of the hash function needs to be at least $2 \log T$ bits (since $2^{(2 \log T)/2} = T$). Taking specific parameters, this means that if we want finding collisions to be as difficult as an exhaustive search over 128-bit keys, then we need the output length of the hash function to be at least 256 bits. We stress that having an output this long is only a *necessary* condition, not a sufficient one. We also note that birthday attacks work only for finding collisions. There are no generic attacks

[1] It can be shown that this is (essentially) the worst case, and collisions occur with higher probability if H deviates from random and the $\{x_i\}$ are chosen uniformly.

for second preimage resistance or preimage resistance of a hash functions H that require fewer than 2^ℓ evaluations of H (though see Section 5.4.3).

Finding meaningful collisions. The birthday attack just described gives a collision that is not necessarily very useful. But the same idea can be used to find "meaningful" collisions as well. Assume Alice wishes to find two messages x and x' such that $H(x) = H(x')$, and furthermore x should be a letter from her employer explaining why she was fired from work, while x' should be a flattering letter of recommendation. (This might allow Alice to forge an appropriate tag on a letter of recommendation if the hash-and-MAC approach is being used by her employer to authenticate messages.) The observation is that the birthday attack only requires the hash inputs $x_1, \ldots, x_q$ to be distinct; they do not need to be random. Alice can carry out a birthday-type attack by generating $q = \Theta(2^{\ell/2})$ messages of the first type and q messages of the second type, and then looking for collisions between messages of the two types. A small change to the analysis from Appendix A.4 shows that this gives a collision between messages of different types with probability roughly 1/2. A little thought shows that it is easy to write the same message in many different ways. For example, consider the following:

> It is *hard/difficult/challenging/impossible* to *imagine/believe* that we will *find/locate/hire* another *employee/person* having similar *abilities/skills/character* as Alice. She has done a *great/super* job.

Any combination of the italicized words is possible, and expresses the same point. Thus, the sentence can be written in $4 \cdot 2 \cdot 3 \cdot 2 \cdot 3 \cdot 2 = 288$ different ways. This is just one sentence and so it is actually easy to generate a message that can be rewritten in 2^{64} different ways—all that is needed are 64 words with one synonym each. Alice can prepare $2^{\ell/2}$ letters explaining why she was fired and another $2^{\ell/2}$ letters of recommendation; with good probability, a collision between the two types of letters will be found.

5.4.2 Small-Space Birthday Attacks

The birthday attacks described above require a large amount of memory; specifically, they require the attacker to store all $\mathcal{O}(q) = \mathcal{O}(2^{\ell/2})$ values $\{y_i\}$, because the attacker does not know in advance which pair of values will yield a collision. This is a significant drawback because memory is, in general, a scarcer resource than time. It is arguably more difficult to allocate and manage storage for 2^{60} bytes than to execute 2^{60} CPU instructions. Furthermore, one can always let a computation run indefinitely, whereas the memory requirements of an algorithm must be satisfied as soon as that amount of memory is needed.

We show here a better birthday attack with drastically reduced memory requirements. In fact, it has similar time complexity and success probability as before, but uses only *constant* memory. The attack begins by choosing a

random value x_0 and then computing $x_i := H(x_{i-1})$ and $x_{2i} := H(H(x_{2(i-1)}))$ for $i = 1, 2, \ldots$. (Note that $x_i = H^{(i)}(x_0)$ for all i, where $H^{(i)}$ refers to i-fold iteration of H.) In each step the values x_i and x_{2i} are compared; if they are equal then there is a collision somewhere in the sequence $x_0, x_1, \ldots, x_{2i-1}$. The algorithm then finds the least value of j for which $x_j = x_{j+i}$ (note that $j \leq i$ since $j = i$ works), and outputs x_{j-1}, x_{j+i-1} as a collision. This attack, described formally as Algorithm 5.9 and analyzed below, only requires storage of two hash values in each iteration.

ALGORITHM 5.9
A small-space birthday attack

Input: A hash function $H : \{0,1\}^* \to \{0,1\}^\ell$
Output: Distinct x, x' with $H(x) = H(x')$

$x_0 \leftarrow \{0,1\}^{\ell+1}$
$x' := x := x_0$
for $i = 1, 2, \ldots$ **do**:
  $x := H(x)$
  $x' := H(H(x'))$
  // now $x = H^{(i)}(x_0)$ and $x' = H^{(2i)}(x_0)$
  if $x = x'$ **break**
$x' := x$, $x := x_0$
for $j = 1$ to i:
  if $H(x) = H(x')$ **return** x, x' and **halt**
  else $x := H(x)$, $x' := H(x')$
  // now $x = H^{(j)}(x_0)$ and $x' = H^{(i+j)}(x_0)$

How many iterations of the first loop do we expect before $x' = x$? Consider the sequence of values $x_1, x_2, \ldots$, where $x_i = H^{(i)}(x_0)$ as defined before. If we model H as a random function, then each of these values is uniformly and independently distributed in $\{0,1\}^\ell$ until the first repeat occurs. Thus, we expect a repeat to occur with probability $1/2$ in the first $q = \Theta(2^{\ell/2})$ terms of the sequence. We show that when there is a repeat in the first q elements, the algorithm finds a repeat in at most q iterations of the first loop:

CLAIM 5.10 *Let $x_1, \ldots, x_q$ be a sequence of values with $x_m = H(x_{m-1})$. If $x_I = x_J$ with $1 \leq I < J \leq q$, then there is an $i < J$ such that $x_i = x_{2i}$.*

PROOF The sequence $x_I, x_{I+1}, \ldots$ repeats with period $\Delta \stackrel{\text{def}}{=} J - I$. That is, for all $i \geq I$ and $k \geq 0$ it holds that $x_i = x_{i+k\cdot\Delta}$. Let i be the smallest multiple of Δ that is also greater than or equal to I. We have $i < J$ since the sequence of Δ values $I, I+1, \ldots I+(\Delta-1) = J-1$ contains a multiple of Δ. Since $i \geq I$ and $2i - i = i$ is a multiple of Δ, it follows that $x_i = x_{2i}$. ∎

Thus, if there is a repeated value in the sequence $x_1, \ldots, x_q$, then there is some $i < q$ for which $x_i = x_{2i}$. But then in iteration i of our algorithm, we have $x = x'$ and the algorithm breaks out of the first loop. At that point in the algorithm, we know that $x_i = x_{i+i}$. The algorithm then sets $x' := x(= x_i)$ and $x := x_0$, and proceeds to find the *smallest* $j \geq 0$ for which $x_j = x_{j+i}$. (Note $j \neq 0$ because $|x_0| = \ell + 1$.) It outputs x_{j-1}, x_{j+i-1} as a collision.

Finding meaningful collisions. The algorithm just described may not seem amenable to finding meaningful collisions since it has no control over the elements sampled. Nevertheless, we show how finding meaningful collisions is possible. The trick is to find a collision in the right function!

Assume, as before, that Alice wishes to find a collision between messages of two different "types," e.g., a letter explaining why Alice was fired and a flattering letter of recommendation that both hash to the same value. Then, Alice writes each message so that there are $\ell - 1$ interchangeable words in each; i.e., there are $2^{\ell-1}$ messages of each type. Define the one-to-one function $g : \{0,1\}^\ell \to \{0,1\}^*$ such that the ℓth bit of the input selects between messages of type 0 or type 1, and the ith bit (for $1 \leq i \leq \ell - 1$) selects between options for the ith interchangeable word in messages of the appropriate type. For example, consider the sentences:

0: Bob is a *good/hardworking* and *honest/trustworthy worker/employee.*
1: Bob is a *difficult/problematic* and *taxing/irritating worker/employee.*

Define a function g that takes 4-bit inputs, where the last bit determines the type of sentence output, and the initial three bits determine the choice of words in that sentence. For example:

$$g(0000) = \text{Bob is a good and honest worker.}$$
$$g(0001) = \text{Bob is a difficult and taxing worker.}$$
$$g(1010) = \text{Bob is a hardworking and honest employee.}$$
$$g(1011) = \text{Bob is a problematic and taxing employee.}$$

Now define $f : \{0,1\}^\ell \to \{0,1\}^\ell$ by $f(x) \stackrel{\text{def}}{=} H(g(x))$. Alice can find a collision in f using the small-space birthday attack shown earlier. The point here is that any collision x, x' in f yields two messages $g(x), g(x')$ that collide under H. If x, x' is a random collision then we expect that with probability $1/2$ the colliding messages $g(x), g(x')$ will be of different types (since x and x' differ in their final bit with that probability). If the colliding messages are not of different types, the process can be repeated again from scratch.

5.4.3 *Time/Space Tradeoffs for Inverting Functions

In this section we consider the question of preimage resistance, i.e., we are interested in algorithms for the problem of function inversion. Here, an algorithm is given $y = H(x)$ for uniform x, and the goal is to find any x' such

that $H(x') = y$. We begin by assuming that the input and output lengths of H are equal, and briefly consider the more general case at the end.

Let $H : \{0,1\}^\ell \to \{0,1\}^\ell$ be a function. Without exploiting any weaknesses of H, finding a preimage of a point y can be done in time $\mathcal{O}(2^\ell)$ via an exhaustive search over the domain. We show that with significant preprocessing, and a relatively large amount of memory, it is possible to do better.

To be clear: we view preprocessing as a one-time operation and we will not be overly concerned with its cost. We are instead interested in the *on-line* time required to invert H at a point y, after the preprocessing has been done. This is justified if the cost of preprocessing can be amortized over the inversion of many points, or if we are willing to invest computational resources for preprocessing before y is known for the benefit of faster inversion afterwards.

In fact, it is trivial to use preprocessing to enable function inversion in very little time. All we need to do is evaluate H on every point during the preprocessing phase, and then store the pairs $\{(x, H(x))\}$ in a table, sorted by their second entry. Upon receiving any point y, a preimage of y can be found easily by searching the table for a pair with second entry y. The drawback here is that we need to allocate space for storing $\mathcal{O}(2^\ell)$ pairs in the table, which can be prohibitive, if not impossible for large ℓ (e.g., $\ell = 80$).

The initial brute-force attack uses constant memory and $\mathcal{O}(2^\ell)$ time, while the attack just described stores $\mathcal{O}(2^\ell)$ points and enables inversion in essentially constant time. We now present an approach that allows an attacker to *trade off* time and memory. Specifically, we show how to store $\mathcal{O}(2^{2\ell/3})$ points and find preimages in time $\mathcal{O}(2^{2\ell/3})$; other tradeoffs are possible.

A warmup. We begin by considering a simple case where the function H defines a cycle, meaning that $x, H(x), H(H(x)), \ldots$ covers all of $\{0,1\}^\ell$ for any starting point x (note that most functions do not define a cycle, but we assume this in order to demonstrate the idea in a very simple case). For clarity, let $N = 2^\ell$ denote the domain size.

In the preprocessing phase, the attacker simply exhausts the entire cycle, beginning at an arbitrary starting point x_0 and computing $x_1 := H(x_0)$, $x_2 := H(H(x_0))$, up to $x_N = H^{(N)}(x_0)$, where $H^{(i)}$ refers to i-fold evaluation of H. Let $x_i \stackrel{\text{def}}{=} H^{(i)}(x_0)$. We imagine partitioning the cycle into $\sqrt{N}$ segments of length $\sqrt{N}$ each, and having the attacker store the points at the beginning and end of each such segment. That is, the attacker stores in a table pairs of the form $(x_{i\cdot\sqrt{N}},\ x_{(i+1)\cdot\sqrt{N}})$, for $i = 0$ to $\sqrt{N} - 1$, sorted by the second component of each pair. The resulting table contains $\mathcal{O}(\sqrt{N})$ points.

When the attacker is given a point y to invert in the on-line phase, it checks which of y, $H(y)$, $H^{(2)}(y)$, ... corresponds to the endpoint of a segment. (Each check just involves a table lookup on the second component of the stored pairs.) Since y lies in some segment, it is guaranteed to hit an endpoint within $\sqrt{N}$ steps. Once an endpoint $x = x_{(i+1)\cdot\sqrt{N}}$ is identified, the attacker takes the starting point $x' = x_{i\cdot\sqrt{N}}$ of the corresponding segment

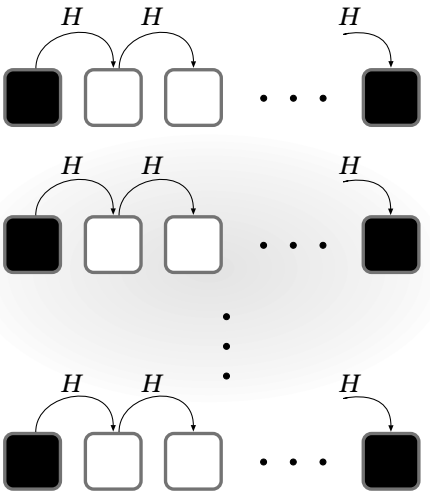


FIGURE 5.3: Table generation. Only the (SP_i, EP_i) pairs are stored.

and computes $H(x')$, $H^{(2)}(x')$, ... until y is reached; this immediately gives the desired preimage. Observe that this takes at most $\sqrt{N}$ evaluations of H.

In summary, this attack stores $\mathcal{O}(\sqrt{N})$ points and finds preimages with probability 1 using $\mathcal{O}(\sqrt{N})$ hash computations.

Hellman's time/space tradeoff. Martin Hellman introduced a more general time/space tradeoff applicable to an arbitrary function H (though the analysis treats H as a random function). Hellman's attack still stores the starting point and endpoint of several segments, but in this case the segments are "independent" rather than being part of one large cycle. In more detail: let s, t be parameters we will set later. The attack first chooses s uniform starting points $SP_1, \ldots, SP_s \in \{0,1\}^\ell$. For each such point SP_i, it computes a corresponding endpoint $EP_i := H^{(t)}(SP_i)$ using t-fold application of H. (See Figure 5.3.) The attacker then stores the values $\{(SP_i, EP_i)\}_{i=1}^s$ in a table, sorted by the second entry of each pair.

Upon receiving a value y to invert, the attack proceeds as in the simple case discussed earlier. Specifically, it checks if any of y, $H(y)$, ..., $H^{(t-1)}(y)$ is equal to the endpoint of some segment (stopping as soon as the first such match is found). It is possible that none of these values is equal to an endpoint (as we discuss below). However, if $H^{(j)}(y) = EP_i = H^{(t)}(SP_i)$ for some i, j, then the attacker computes $H^{(t-j-1)}(SP_i)$ and checks whether this is a preimage of y. The entire process requires at most t evaluations of H.

This seems to work, but there are several subtleties we have ignored. First, it may happen that none of y, $H(y)$, ..., $H^{(t-1)}(y)$ is the endpoint of a segment. This can happen if y is not in the collection of $s \cdot t$ values (not counting the starting points) obtained during the initial process of generating the table. We can set $s \cdot t \geq N$ in an attempt to include every ℓ-bit string in the

table, but this does not solve the problem since there can be collisions in the table itself—in fact, for $s \cdot t \geq N^{1/2}$ our previous analysis of the birthday problem tells us that collisions are likely—which will reduce the number of distinct points in the collection of values. A second problem, which arises even if y *is* in the table, is that even if we find a matching endpoint, and so $H^{(j)}(y) = EP_i = H^{(t)}(SP_i)$ for some i, j, this does not guarantee that $H^{(t-j-1)}(SP_i)$ is a preimage of y. The issue here is that the segment $y, H(y), \ldots, H^{(t-1)}(y)$ might collide with the ith segment even though y itself is not in that segment; see Figure 5.4. (Even if y lies in some segment, the first matching endpoint may not be in that segment.) We call this a *false positive*. One might think this is unlikely to occur if H is collision resistant; again, however, we are dealing with a situation where more than $\sqrt{N}$ points are involved and so collisions actually become likely.

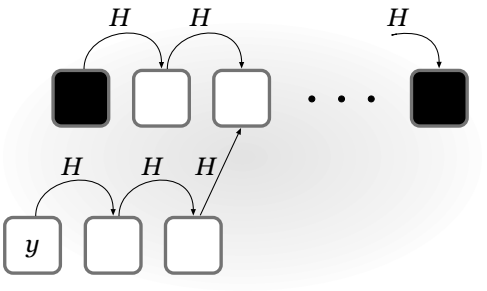


FIGURE 5.4: Colliding in the on-line phase.

The problem of false positives can be addressed by modifying the algorithm so that it always computes the entire sequence $y, H(y), \ldots, H^{(t-1)}(y)$, and checks whether $H^{(t-j-1)}(SP_i)$ is a preimage of y for *every* i, j such that $H^{(j)}(y) = EP_i$. This is guaranteed to find a preimage as long as y is in the collection of values (not including the starting points) generated during preprocessing. A concern now is that the running time of the algorithm might increase, since each false positive incurs an additional $\mathcal{O}(t)$ hash evaluations. One can show that the expected number of false positives is $\mathcal{O}(st^2/N)$. (There are t values in the sequence $y, H(y), \ldots, H^{(t-1)}(y)$ and at most st distinct points in the table. Treating H as a random function, the probability that any point in the sequence equals some point in the table is $1/N$. The expected number of false positives is thus $t \cdot st \cdot 1/N = st^2/N$.) Thus, as long as $st^2 \approx N$, which we will ensure for other reasons below, the expected number of false positives is constant and dealing with false positives is expected to require only $\mathcal{O}(t)$ additional hash computations.

Given the above modification, the probability of inverting $y = H(x)$ is at least the probability that x is in the collection of points (not including the endpoints) generated during preprocessing. We now lower bound this probability, taken over the randomness of the preprocessing stage as well as

uniform choice of x, treating H as a random function in the analysis. We first compute the expected number of distinct points in the table. Consider what happens when the ith row of the table is generated. The starting point SP_i is uniform and there are at most $(i-1)\cdot t$ distinct points (not including the endpoints) in the table already, so the probability that SP_i is "new" (i.e., not equal to any previous value) is at least $1-(i-1)\cdot t/N$. What is the probability that $H(SP_i)$ is new? If SP_i is not new, then almost surely neither is $H(SP_i)$. On the other hand, if SP_i *is* new then $H(SP_i)$ is uniform (because we treat H as a random function) and so is new with probability at least $1-((i-1)\cdot t+1)/N$. (We now have the additional point SP_i.) Thus, the probability that $H(SP_i)$ is new is at least

$$\begin{aligned}\Pr[SP_i \text{ is new}] \cdot \Pr[H(SP_i) \text{ is new} \mid SP_i \text{ is new}] \\ \geq \left(1-\frac{(i-1)\cdot t}{N}\right)\cdot\left(1-\frac{(i-1)\cdot t+1}{N}\right) \\ > \left(1-\frac{(i-1)\cdot t+1}{N}\right)^2.\end{aligned}$$

Continuing in this way, the probability that $H^{(t-1)}(SP_i)$ is new is at least

$$\left(1-\frac{i\cdot t}{N}\right)^t = \left[\left(1-\frac{i\cdot t}{N}\right)^{\frac{N}{i\cdot t}}\right]^{\frac{i\cdot t^2}{N}} \approx e^{-it^2/N}.$$

The thing to notice here is that when $it^2 \leq N/2$, this probability is at least $1/2$; on the other hand, once $it^2 > N$ the probability is rather small. Considering the last row, when $i = s$, this means that we will not gain much additional coverage if $st^2 > N$. A good setting of the parameters is thus $st^2 = N/2$. Assuming this, the expected number of distinct points in the table is

$$\sum_{i=1}^{s}\sum_{j=0}^{t-1}\Pr\left[H^{(j)}(SP_i) \text{ is new}\right] \geq \sum_{i=1}^{s}\sum_{j=0}^{t-1}\frac{1}{2} = \frac{st}{2}.$$

The probability that x is "covered" is then at least $\frac{st}{2N} = \frac{1}{4t}$.

This gives a weak time/space tradeoff, in which we can use more space (and consequently less time) at the expense of decreasing the probability of inverting y. But we can do better by generating $T = 4t$ "independent" tables. (This increases both the space and time by a factor of T.) As long as we can treat the probabilities of x being in each of the associated tables as independent, the probability that *at least one* of these tables contains x is

$$1-\Pr[\text{no table contains } x] = 1-\left(1-\frac{1}{4t}\right)^{4t} \approx 1-e^{-1} = 0.63.$$

The only remaining question is how to generate an independent table. (Note that generating a table exactly as before is the same as adding s additional

rows to our original table, which we have already seen does not help.) We can do this for the ith such table by applying some function F_i after every evaluation of H, where $F_1, \ldots, F_T$ are all distinct. (A good choice might be to set $F_i(x) = x \oplus c_i$ for some fixed constant c_i that is different in each table.) Let $H_i \stackrel{\text{def}}{=} F_i \circ H$, i.e., $H_i(x) = F_i(H(x))$. Then for the ith table we again choose s random starting points, but for each such point we now compute $H_i(SP), H_i^{(2)}(SP)$, and so on. Upon receiving a value $y = H(x)$ to invert, the attacker first computes $y' = F_i(y)$ and then checks if any of y', $H_i(y')$, $\ldots$, $H_i^{(t-1)}(y')$ corresponds to an endpoint in the ith table; this is repeated for $i = 1, \ldots, T$. (We omit further details.) While it is difficult to argue independence formally, this approach leads to good results in practice.

Choosing parameters. Summarizing the above discussion, we see that as long as $st^2 = N/2$ we have an algorithm that stores $\mathcal{O}(s \cdot T) = \mathcal{O}(s \cdot t) = \mathcal{O}(N/t)$ points during a preprocessing phase, and can then invert y with constant probability in time $\mathcal{O}(t \cdot T) = \mathcal{O}(t^2)$. One setting of the parameters is $t = N^{1/3} = 2^{\ell/3}$, in which case we have an algorithm storing $\mathcal{O}(2^{2\ell/3})$ points that finds preimages using $\mathcal{O}(2^{2\ell/3})$ hash computations. If a hash function with 80 bits of output is used, then this is feasible in practice.

Handling different domain and range. In practice, it is common to be faced with a situation in which the domain and range of H are different. One example is in the context of password cracking (see Section 5.6.3), where an attacker has $H(pw)$ but $|pw| \ll \ell$. In the general case, say x is chosen from some domain D which may be larger or smaller than $\{0,1\}^\ell$. While it is, of course, possible to artificially expand the domain/range to make them match, this will *not* be useful for the attack described above. To see why, consider the password example. For the attack to succeed we want pw to be in some table of values generated during preprocessing. If we generate each row of the table by simply computing $H(SP), H^{(2)}(SP), \ldots$, for $SP \in D$, then *none* of these values (except possibly SP itself) will be equal to pw.

We can address this by applying a function F_i, as before, between each evaluation of H, though now we choose F_i mapping $\{0,1\}^\ell$ to D. This solves the above issue, since $F_i(H(SP)), (F_i \circ H)^{(2)}(SP), \ldots$ now all lie in D.

Applications to key-recovery attacks. Time/space tradeoffs give attacks on cryptographic primitives other than hash functions. One canonical application—in fact, the application originally considered by Hellman—is an attack on an arbitrary block cipher F that leads to recovery of the key. Define $H(k) \stackrel{\text{def}}{=} F_k(x)$ where x is some arbitrary, but fixed, input that will be used for building the table. If an attacker can obtain $F_k(x)$ for an unknown key k—either via a chosen-plaintext attack or by choosing x such that $F_k(x)$ is likely to be obtained in a known-plaintext attack—then by inverting H the attacker learns (a candidate value for) k. Note that it is possible for the key length of F to differ from its block length, but in this case we can use the technique just described for handling H with different domain and range.

5.5 The Random-Oracle Model

There are several examples of constructions based on cryptographic hash functions that cannot be proven secure based only on the assumption that the hash function is collision or preimage resistant. (We will see some in the following section.) In many cases, there appears to be *no* simple and reasonable assumption regarding the hash function that would be sufficient for proving the construction secure.

Faced with this situation, there are several options. One is to look for schemes that *can* be proven secure based on some reasonable assumption about the underlying hash function. This is a good approach, but it leaves open the question of what to do until such schemes are found. Also, provably secure constructions may be significantly less efficient than other approaches that have not been proven secure. (This is a prominent issue that we will encounter in the setting of public-key cryptography.)

Another possibility, of course, is to use an existing cryptosystem even if it has no justification for its security other than, perhaps, the fact that the designers tried to attack it and were unsuccessful. This flies in the face of everything we have said about the importance of the rigorous, modern approach to cryptography, and it should be clear that this is unacceptable.

An approach that has been hugely successful in practice, and which 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. Although 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 treats a cryptographic hash function H as a truly random function. (We have already seen an example of this in our discussion of time/space tradeoffs, although there we were analyzing an attack rather than a construction.) More specifically, the random-oracle model posits the existence of a public, random function H that can be evaluated *only* by "querying" an oracle—which can be thought of as a "black box"—that returns $H(x)$ when given input x. (We will discuss how this is to be interpreted in the following section.) To differentiate things, the model we have been using until now (where no random oracle is present) is often called the "standard model."

No one claims that a random oracle exists, although there have been suggestions that a random oracle could be implemented in practice using a trusted party (i.e., some server on the Internet). Rather, the random-oracle model provides a formal *methodology* that can be used to design and validate cryptographic schemes using the following two-step approach:

1. First, a scheme is designed and proven secure in the random-oracle model. That is, we assume the world contains a random oracle, and construct and analyze a cryptographic scheme within this model. 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 is *instantiated* with an appropriately designed cryptographic hash function $\hat{H}$. (We return to this point at the end of this section.) That is, at each point where the scheme dictates that a party should query the oracle for the value $H(x)$, the party instead computes $\hat{H}(x)$ on its own.

The hope is that the cryptographic hash function used in the second step is "sufficiently good" at emulating a random oracle, so that the security proof given in the first step will carry over to the real-world instantiation of the scheme. The difficulty here is that there is no theoretical justification for this hope, and in fact there are (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, it is not clear (mathematically or heuristically) what it means for a hash function to be "sufficiently good" at emulating a random oracle, nor is it clear that this is an achievable goal. In particular, no concrete instantiation $\hat{H}$ can ever behave like a random function, since $\hat{H}$ is deterministic and fixed. For these reasons, a proof of security in the random-oracle model should be viewed as providing evidence that a scheme has no "inherent design flaws," but is *not* 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 5.5.2.

5.5.1 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—honest parties as well as the adversary—can interact with the box, where such interaction consists of feeding in a binary string x as input and receiving a binary string y as output; we refer to this as *querying the oracle on* x, and call x itself a *query* made to the oracle. Queries to the oracle are assumed to be private so that if some party queries the oracle on input x then no one else learns x, or even learns that this party queried the oracle at all. This makes sense, because calls to the oracle correspond (in the real-world instantiation) to local evaluations of a cryptographic hash function.

An important property of this "box" is that it 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 well-defined function H; i.e., we define the function H in terms of the input/output characteristics of the box. For convenience, we thus speak of "querying H" rather than querying the box. No one "knows" the entire function H (except the box itself); at best, all that is known are the values of H on the strings that have been explicitly queried thus far.

We have already discussed in Chapter 3 what it means to choose a random function H. We only reiterate here that there are two equivalent ways to think about the uniform selection of H: either picture H being chosen "in one shot" uniformly from the set of all functions on some specified domain and range, or imagine generating outputs for H "on-the-fly," as needed. Specifically, in the second case we can view the function as being defined by a table 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 value y_i is returned. Otherwise, a *uniform* string $y \in \{0,1\}^\ell$ is chosen (for some specified ℓ), the answer y is returned, and the oracle stores (x, y) in its table. This second viewpoint is often conceptually easier to reason about, and is also technically easier to deal with if H is defined over an infinite domain (e.g., $\{0,1\}^*$).

When we defined pseudorandom functions in Section 3.5.1, we also considered algorithms having oracle access to a random function. Lest there be any confusion, we note that the usage of a random function there is very different from the usage of a random function here. There, a random function was used *as a way of defining* what it means for a (concrete) keyed function to be pseudorandom. In the random-oracle model, in contrast, the random function is used *as part of a construction itself* and must somehow be instantiated in the real world if we want a concrete realization of the construction. A pseudorandom function is not a random oracle because it is only pseudorandom if the key is *secret*. However, in the random-oracle model all parties need to be able to compute the function; thus there can be no secret key.

Definitions and Proofs in the Random-Oracle Model

Definitions in the random-oracle model are slightly different from their counterparts in the standard model because the probability spaces considered in each case are not the same. In the standard model a scheme Π is secure if for all PPT adversaries $\mathcal{A}$ the probability of some event is below some threshold, where *this probability is taken over the random choices of the parties running* Π *and those of the adversary* $\mathcal{A}$. Assuming the honest parties who use Π in the real world make random choices as directed by the scheme, satisfying a definition of this sort guarantees security for real-world usage of Π.

In the random-oracle model, in contrast, a scheme Π may rely on an oracle H. As before, Π is secure if for all PPT adversaries $\mathcal{A}$ the probability of some event is below some threshold, but now *this probability is taken over random choice of* H as well as the random choices of the parties running Π and those of the adversary $\mathcal{A}$. When using Π in the real world, some (instantiation of) H must be fixed. Unfortunately, security of Π is not guaranteed

for any *particular* choice of H. This indicates one reason why it is difficult to argue that any concrete instantiation of the oracle H by a deterministic function yields a secure scheme. (An additional, technical, difficulty is that once a concrete function H is fixed, the adversary $\mathcal{A}$ is no longer restricted to querying H as an oracle but can instead look at and use the *code* of H in the course of its attack.)

Proofs in the random-oracle model can exploit the fact that H is chosen at random, and that the only way to evaluate $H(x)$ is to explicitly query x to H. Three properties in particular are especially useful; we sketch them informally here, and show some simple applications of them below and in the next section, but caution that a full understanding will likely have to wait until we present formal proofs in the random-oracle model in later chapters.

The first useful property of the random-oracle model is:

> *If x has not been queried to H, then the value of $H(x)$ is* **uniform**.

This may seem superficially similar to the guarantee provided by a pseudorandom generator, but is actually much stronger. If G is a pseudorandom generator then $G(x)$ is pseudorandom to an observer *assuming x is chosen uniformly at random and is completely unknown to the observer.* If H is a random oracle, however, then $H(x)$ is truly uniform to an observer as long as the observer has not queried x. This is true even if x is known, or if x is not uniform but *is* hard to guess. (For example, if x is an n-bit string where the first half of x is known and the last half is random then $G(x)$ might be easy to distinguish from random but $H(x)$ will not be.)

The remaining two properties relate explicitly to *proofs by reduction* in the random-oracle model. (It may be helpful here to review Section 3.3.2.) As part of the reduction, the random oracle that the adversary $\mathcal{A}$ interacts with must be simulated. That is: $\mathcal{A}$ will submit queries to, and receive answers from, what it believes to be the oracle, but the reduction itself must now answer these queries. This turns out to give a lot of power. For starters:

> *If $\mathcal{A}$ queries x to H, the reduction can* **see this query** *and learn x.*

This is sometimes called "extractability." (This does not contradict the fact, mentioned earlier, that queries to the random oracle are "private." While that is true in the random-oracle model itself, here we are using $\mathcal{A}$ as a subroutine within a reduction that is simulating the random oracle for $\mathcal{A}$.) Finally:

> *The reduction can* **set** *the value of $H(x)$ (i.e., the response to query x) to a value of its choice, as long as this value is correctly distributed, i.e., uniform.*

This is called "programmability." There is no counterpart to extractability or programmability once H is instantiated with any concrete function.

Simple Illustrations of the Random-Oracle Model

At this point some examples may be helpful. The examples given here are relatively simple, and do not use the full power that the random-oracle model affords. Rather, these examples are presented merely to provide a gentle introduction to the model. In what follows, we assume a random oracle mapping ℓ_{in}-bit inputs to ℓ_{out}-bit outputs, where $\ell_{in}, \ell_{out} > n$, the security parameter (so ℓ_{in}, ℓ_{out} are functions of n).

A random oracle as a pseudorandom generator. We first show that, for $\ell_{out} > \ell_{in}$, a random oracle can be used as a pseudorandom generator. (We do not say that a random oracle *is* a pseudorandom generator, since a random oracle is not a fixed function.) Formally, we claim that for any PPT adversary $\mathcal{A}$, there is a negligible function negl such that

$$\left| \Pr[\mathcal{A}^{H(\cdot)}(y) = 1] - \Pr[\mathcal{A}^{H(\cdot)}(H(x)) = 1] \right| \leq \mathsf{negl}(n),$$

where in the first case the probability is taken over uniform choice of H, uniform choice of $y \in \{0,1\}^{\ell_{out}(n)}$, and the randomness of $\mathcal{A}$, and in the second case the probability is taken over uniform choice of H, uniform choice of $x \in \{0,1\}^{\ell_{in}(n)}$, and the randomness of $\mathcal{A}$. We have explicitly indicated that $\mathcal{A}$ has oracle access to H in each case; once H has been chosen then $\mathcal{A}$ can freely make queries to it.

Let S denote the set of points on which $\mathcal{A}$ has queried H; of course, $|S|$ is polynomial in n. Observe that in the second case, the probability that $x \in S$ is negligible. This holds since $\mathcal{A}$ starts with no information about x (note that $H(x)$ by itself reveals nothing about x because H is a random function), and because S is exponentially smaller than $\{0,1\}^{\ell_{in}}$. Moreover, conditioned on $x \notin S$ in the second case, $\mathcal{A}$'s input in each case is a uniform string that is independent of the answers to $\mathcal{A}$'s queries.

A random oracle as a collision-resistant hash function. If $\ell_{out} < \ell_{in}$, a random oracle is collision resistant. That is, the success probability of any PPT adversary $\mathcal{A}$ in the following experiment is negligible:

1. A random function H is chosen.
2. $\mathcal{A}$ succeeds if it outputs distinct x, x' with $H(x) = H(x')$.

To see this, assume without loss of generality that $\mathcal{A}$ only outputs values x, x' that it had previously queried to the oracle, and that $\mathcal{A}$ never makes the same query to the oracle twice. Letting the oracle queries of $\mathcal{A}$ be $x_1, \ldots, x_q$, with $q = \mathsf{poly}(n)$, it is clear that the probability that $\mathcal{A}$ succeeds is upper-bounded by the probability that $H(x_i) = H(x_j)$ for some $i \neq j$. But this is exactly equal to the probability that if we pick q strings $y_1, \ldots, y_q \in \{0,1\}^{\ell_{out}}$ independently and uniformly at random, we have $y_i = y_j$ for some $i \neq j$. This is exactly the birthday problem, and so using the results of Appendix A.4 we have that $\mathcal{A}$ succeeds with negligible probability $\mathcal{O}(q^2/2^{\ell_{out}})$.

Constructing a pseudorandom function from a random oracle. It is also rather easy to construct a pseudorandom function in the random-oracle model. Suppose $\ell_{in}(n) = 2n$ and $\ell_{out}(n) = n$, and define

$$F_k(x) \stackrel{\text{def}}{=} H(k\|x),$$

where $|k| = |x| = n$. In Exercise 5.11 you are asked to show that this is a pseudorandom function, namely, that for any polynomial-time $\mathcal{A}$ the success probability of $\mathcal{A}$ in the following experiment is not more than $1/2$ plus a negligible function:

1. A function H and values $k \in \{0,1\}^n$ and $b \in \{0,1\}$ are chosen uniformly.
2. If $b = 0$, the adversary $\mathcal{A}$ is given access to an oracle for $F_k(\cdot) = H(k\|\cdot)$. If $b = 1$, then $\mathcal{A}$ is given access to a random function mapping n-bit inputs to n-bit outputs. (This random function is *independent* of H.)
3. $\mathcal{A}$ outputs a bit b', and succeeds if $b' = b$.

In step 2, $\mathcal{A}$ can access H in addition to the function oracle provided to it by the experiment. (A pseudorandom function in the random-oracle model must be indistinguishable from a random function that is independent of H.)

An interesting aspect of all the above claims is that they make no computational assumptions; they hold even for computationally unbounded adversaries as long as those adversaries are limited to making polynomially many queries to the oracle. This has no counterpart in the real world, where we have seen that computational assumptions are necessary.

5.5.2 Is the Random-Oracle Methodology Sound?

Schemes designed in the random-oracle model are implemented in the real world by instantiating H with some concrete function. With the mechanics of the random-oracle model behind us, we turn to a more fundamental question:

> *What do proofs of security in the random-oracle model guarantee as far as security of any real-world instantiation?*

This question does not have a definitive answer: there is currently debate within the cryptographic community regarding how to interpret proofs in the random-oracle model, and an active area of research is to determine what, precisely, a proof of security in the random-oracle model implies vis-a-vis the real world. We can only hope to give a flavor of both sides of the debate.

Objections to the random-oracle model. The starting point for arguments against using random oracles is simple: as we have already noted, there is no formal or rigorous justification for believing that a proof of security for some scheme Π in the random-oracle model says anything about the security of Π in the real world, once the random oracle H has been instantiated with

any particular hash function $\hat{H}$. This is more than just theoretical uneasiness. A little thought shows that *no* concrete hash function can ever act as a "true" random oracle. For example, in the random-oracle model the value $H(x)$ is "completely random" if x was not explicitly queried. The counterpart would be to require that $\hat{H}(x)$ is random (or pseudorandom) if $\hat{H}$ was not explicitly evaluated on x. How are we to interpret this in the real world? 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 does not 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 determined.

Limitations of the random-oracle model become clearer once we examine the proof techniques introduced earlier. 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. If we replace the random oracle by a particular hash function $\hat{H}$, this means 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 previously, the notion of $\mathcal{A}$ performing explicit evaluations of $\hat{H}$ may not be true and certainly cannot be formally defined.) Likewise, proofs of security in the random-oracle model allow the reduction to choose the outputs of H as it wishes, something that is clearly not possible when a concrete function is used.

Even if we are willing to overlook the above theoretical concerns, a practical problem is that we do not currently have a very good understanding of what it means for a concrete hash function to be "sufficiently good" at instantiating a random oracle. For concreteness, say we want to instantiate the random oracle using some appropriate modification of SHA-1 (SHA-1 is a cryptographic hash function discussed in Section 6.3.3). While for some particular scheme Π it might be reasonable to assume that Π is secure when instantiated using SHA-1, it is much less reasonable to assume that SHA-1 can take the place of a random oracle in *every* scheme designed in the random-oracle model. Indeed, as we have said earlier, we *know* that SHA-1 is not a random oracle. And it is not hard to design a scheme that is secure in the random-oracle model, but is insecure when the random oracle is replaced by SHA-1.

We emphasize that an assumption of the form "SHA-1 acts like a random oracle" is qualitatively different from assumptions such as "SHA-1 is collision resistant" or "AES is a pseudorandom function." The problem lies partly with the fact that there is no satisfactory *definition* of what the first statement means, while we do have such definitions for the latter two statements.

Because of this, using the random-oracle model to prove security of a scheme is *qualitatively* different from, e.g., introducing a new cryptographic assumption in order to prove a scheme secure in the standard model. Therefore, proofs of security in the random-oracle model are less satisfying than proofs of security in the standard model.

Support for the random-oracle model. Given all the problems with the random-oracle model, why use it at all? More to the point: why has the random-oracle model been so influential in the development of modern cryptography (especially current practical usage of cryptography), and why does it continue to be so widely used? As we will see, the random-oracle model 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 deployed schemes having proofs of security in the random-oracle model. In addition, proofs in the random-oracle model are almost universally recognized as lending confidence to the security of schemes being considered for standardization.

The fundamental reason for this is the belief that:

> *A proof of security in the random-oracle model is significantly better than no proof at all.*

Although some disagree, we offer the following in support of this assertion:

- A proof of security for a scheme in the random-oracle model indicates that the scheme's design is "sound," in the sense that the only possible attacks on 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. Thus, if a "good enough" hash function is used to instantiate the random oracle, we should have confidence in the security of the scheme. Moreover, if a given instantiation of the scheme *is* successfully attacked, we can simply replace the hash function being used with a "better" one.
- Importantly, *there have been no successful real-world attacks on schemes proven secure in the random-oracle model*, when the random oracle was instantiated properly. (We do not include here attacks on "contrived" schemes, but remark that great care must be taken in instantiating the random oracle, as indicated by Exercise 5.10.) This gives evidence to the usefulness of the random-oracle model in designing practical schemes.

Nevertheless, the above ultimately represent only intuitive speculation as to the usefulness of proofs in the random-oracle model and—all else being equal—proofs without random oracles are preferable.

Instantiating the Random Oracle

Properly instantiating a random oracle is subtle, and a full discussion is beyond the scope of this book. Here we only alert the reader that using an "off-the-shelf" cryptographic hash function without modification is not, generally speaking, a sound approach. For one thing, most cryptographic hash functions are constructed using the Merkle–Damgård paradigm (cf. Section 5.2), which can be distinguished easily from a random oracle when variable-length inputs

are allowed. (See Exercise 5.10.) Also, in some constructions it is necessary for the output of the random oracle to lie in a certain range (e.g., the oracle should output elements of some group), which results in additional complications.

5.6 Additional Applications of Hash Functions

We conclude this chapter with a brief discussion of some additional applications of cryptographic hash functions in cryptography and computer security.

5.6.1 Fingerprinting and Deduplication

When using a collision-resistant hash function H, the hash (or *digest*) of a file serves as a unique identifier for that file. (If any other file is found to have the same identifier, this implies a collision in H). The hash $H(x)$ of a file x is like a fingerprint, and one can check whether two files are equal by comparing their digests. This simple idea has many applications.

- *Virus fingerprinting:* Virus scanners identify viruses and block or quarantine them. One of the most basic steps toward this goal is to store a database containing the hashes of known viruses, and then to look up the hash of a downloaded application or email attachment in this database. Since only a short string needs to be recorded (and/or distributed) for each virus, the overhead involved is feasible.

- *Deduplication:* Data deduplication is used to eliminate duplicate copies of data, especially in the context of cloud storage where multiple users rely on a single cloud service to store their data. The observation here is that if multiple users wish to store the same file (e.g., a popular video), then the file only needs to be stored once and need not be uploaded separately by each user. Deduplication can be achieved by first having a user upload a hash of the new file they want to store; if a file with this hash is already stored in the cloud, then the cloud-storage provider can simply add a pointer to the existing file to indicate that this specific user has also stored this file. This saves both communication and storage, and the soundness of the methodology follows from the collision resistance of the hash function.

- *Peer-to-peer (P2P) file sharing*: In P2P file-sharing systems, tables are held by servers to provide a file-lookup service. These tables contain the hashes of the available files, once again providing a unique identifier without using much memory.

It may be surprising that a small digest can uniquely identify every file in the world. But this is the guarantee provided by collision-resistant hash functions, which makes them useful in the settings above.

5.6.2 Merkle Trees

Consider a client who uploads a file x to a server. When the client later retrieves x, it wants to make sure that the server returns the original, unmodified file. The client could simply store x and check that the retrieved file is equal to x, but that defeats the purpose of using the server in the first place. We are looking for a solution in which the storage of the client is small.

A natural solution is to use the "fingerprinting" approach described above. The client can locally store the short digest $h := H(x)$; when the server returns a candidate file x' the client need only check that $H(x') \stackrel{?}{=} h$.

What happens if we want to extend this solution to *multiple* files $x_1, \ldots, x_t$? There are two obvious ways of doing this. One is to simply hash each file independently; the client will locally store the digests $h_1, \ldots, h_t$, and verify retrieved files as before. This has the disadvantage that the client's storage grows linearly in t. Another possibility is to hash all the files together. That is, the client can compute $h := H(x_1, \ldots, x_t)$ and store only h. The drawback now is that when the client wants to retrieve and verify correctness of the ith file x_i, it needs to retrieve *all* the files in order to recompute the digest.

Merkle trees, introduced by Ralph Merkle, give a tradeoff between these extremes. A Merkle tree computed over input values $x_1, \ldots, x_t$ is simply a binary tree of depth $\log t$ in which the inputs are placed at the leaves, and the value of each internal node is the hash of the values of its two children; see Figure 5.5. (We assume t is a power of 2; if not, then we can fix some input values to null or use an incomplete binary tree, depending on the application.)

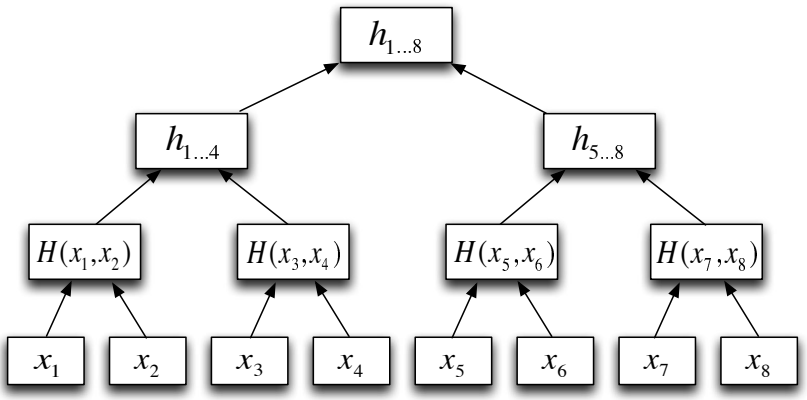


FIGURE 5.5: A Merkle tree.

Fixing some hash function H, we denote by $\mathcal{MT}_t$ the function that takes t input values $x_1, \ldots, x_t$, computes the resulting Merkle tree, and outputs the value of the root of the tree. (A keyed hash function yields a keyed function $\mathcal{MT}_t$ in the obvious way.) We have:

THEOREM 5.11 *Let* (Gen_H, H) *be collision resistant. Then* $(\mathsf{Gen}_H, \mathcal{MT}_t)$ *is also collision resistant for any fixed* t*.*

Merkle trees thus provide an alternative to the Merkle–Damgård transform for achieving domain extension for collision-resistant hash functions. (As described, however, Merkle trees are *not* collision resistant if the number of input values t is allowed to vary.)

Merkle trees provide an efficient solution to our original problem, since they allow verification of any of the original t inputs using $\mathcal{O}(\log t)$ communication. The client computes $h := \mathcal{MT}_t(x_1, \ldots, x_t)$, uploads $x_1, \ldots, x_t$ to the server, and stores h (along with the number of files t) locally. When the client retrieves the ith file, the server sends x_i along with a "proof" π_i that this is the correct value. This proof consists of the values of the nodes in the Merkle tree adjacent to the path from x_i to the root. From these values the client can recompute the value of the root and verify that it is equal to the stored value h. As an example, consider the Merkle tree in Figure 5.5. The client computes $h_{1\ldots8} := \mathcal{MT}_8(x_1, \ldots, x_8)$, uploads $x_1, \ldots, x_8$ to the server, and stores $h_{1\ldots8}$ locally. When the client retrieves x_3, the server sends x_3 along with x_4, $h_{1\ldots2} = H(x_1, x_2)$, and $h_{5\ldots8} = H(H(x_5, x_6), H(x_7, x_8))$. (If files are large we may wish to avoid sending any file other than the one the client has requested. That can easily be done if we define the Merkle tree over the *hashes* of the files rather than the files themselves. We omit the details.) The client computes $h'_{1\ldots4} := H(h_{1\ldots2}, H(x_3, x_4))$ and $h'_{1\ldots8} := H(h'_{1\ldots4}, h_{5\ldots8})$, and then verifies that $h_{1\ldots8} \stackrel{?}{=} h'_{1\ldots8}$.

If H is collision resistant, it is infeasible for the server to send an incorrect file (and any proof) that will cause verification to succeed. Using this approach, the client's local storage is *constant* (independent of the number of files t), and the communication from server to client is proportional to $\log t$.

5.6.3 Password Hashing

One of the most common and important uses of hash functions in computer security is for password protection. Consider a user typing in a password before using their laptop. To authenticate the user, some form of the user's password must be stored somewhere on their laptop. If the user's password is stored in the clear, then an adversary who steals the laptop can read the user's password off the hard drive and then login as that user. (It may seem pointless to try to hide one's password from an attacker who can already read the contents of the hard drive. However, files on the hard drive may be

encrypted with a key derived from the user's password, and would thus only be accessible after the password is entered. In addition, the user is likely to use the same password at other sites.)

This risk can be mitigated by storing a *hash of the password* instead of the password itself. That is, the hard drive stores the value $hpw = H(pw)$ in a password file; later, when the user enters its password pw, the operating system checks whether $H(pw) \stackrel{?}{=} hpw$ before granting access. The same basic approach is also used for password-based authentication on the web. Now, if an attacker steals the hard drive (or breaks into a web server), all it obtains is the hash of the password and not the password itself.

If the password is chosen from some relatively small space D of possibilities (e.g., D might be a dictionary of English words, in which case $|D| \approx 80,000$), an attacker can enumerate all possible passwords $pw_1, pw_2, \ldots \in D$ and, for each candidate pw_i, check whether $H(pw_i) = hpw$. We would like to claim that an attacker can do no better than this. (This would also ensure that the adversary could not learn the password of any user who chose a *strong* password from a large space.) Unfortunately, preimage resistance (i.e., one-wayness) of H is not sufficient to imply what we want. For one thing, preimage resistance only says that $H(x)$ is hard to invert when x is chosen uniformly from a large domain like $\{0,1\}^n$. It says nothing about the hardness of inverting H when x is chosen from some other space, or when x is chosen according to some other distribution. Moreover, preimage resistance says nothing about the *concrete* amount of time needed to find a preimage. For example, a hash function H for which computing $x \in \{0,1\}^n$ given $H(x)$ requires time $2^{n/2}$ could still qualify as preimage resistant, yet this would mean that a 30-bit password could be recovered in only 2^{15} time.

If we model H as a random oracle, then we can formally prove the security we want, namely, recovering pw from hpw (assuming pw is chosen uniformly from D) requires $|D|/2$ evaluations of H, on average.

The above discussion assumes no preprocessing is done by the attacker. As we have seen in Section 5.4.3, though, preprocessing can be used to generate large tables that enable inversion (even of a random function!) faster than exhaustive search. This is a significant concern in practice: even if a user chooses their password as a random combination of 8 alphanumeric characters—giving a password space of size $N = 62^8 \approx 2^{47.6}$—there is an attack using time and space $N^{2/3} \approx 2^{32}$ that will be highly effective. The tables only need to be generated once, and can be used to crack hundreds of thousands of passwords in case of a server breach. Such attacks are routinely carried out in practice.

Mitigation. We briefly describe two mechanisms used to mitigate the threat of password cracking; further discussion can be found in texts on computer security. One technique is to use "slow" hash functions, or to slow down existing hash functions by using multiple iterations (i.e., computing $H^{(I)}(pw)$ for $I \gg 1$). This has the effect of slowing down legitimate users by a factor of I, which is not a problem if I is set to some "moderate" value (e.g., 1,000).

On the other hand, it has a significant impact on an adversary attempting to crack thousands of passwords at once.

A second mechanism is to introduce a *salt*. When a user registers their password, the laptop/server will generate a long random value s (a "salt") unique to that user, and store $(s, hpw = H(s, pw))$ instead of merely storing $H(pw)$ as before. Since s is unknown to the attacker in advance, preprocessing is ineffective and the best an attacker can do is to wait until it obtains the password file and then do a linear-time exhaustive search over the domain D as discussed before. Note also that since a different salt is used for each stored password, a separate brute-force search is needed to recover each password.

5.6.4 Key Derivation

All the symmetric-key cryptosystems we have seen require a *uniformly distributed* bit-string for the secret key. Often, however, it is more convenient for two parties to rely on shared information such as a password or biometric data that is *not* uniformly distributed. (Jumping ahead, in Chapter 10 we will see how parties can interact to generate a high-entropy shared secret that is *not* uniformly distributed.) The parties could try to use their shared information directly as a secret key, but in general this will not be secure (since, e.g., private-key schemes all assume a uniformly distributed key). Moreover, the shared data may not even have the correct format to be used as a secret key (it may be too long, for example).

Truncating the shared secret, or mapping it in some other ad hoc way to a string of the correct length, may lose a significant amount of entropy. (We define one notion of entropy more formally below, but for now one can think of entropy as the logarithm of the space of possible shared secrets.) For example, imagine two parties share a password composed of 28 random uppercase letters, and want to use a cryptosystem with a 128-bit key. Since there are 26 possibilities for each character, there are $26^{28} > 2^{130}$ possible passwords. If the password is shared in ASCII format, each character is stored using 8 bits, and so the total length of the password is 224 bits. If the parties truncate their password to the first 128 bits, they will be using only the first 16 characters of their password. However, this will not be a uniformly distributed 128-bit string! In fact, the ASCII representations of the letters A–Z lie between 0x41 and 0x5A; in particular, the first 3 bits of every byte are always 010. This means that *37.5% of the bits of the resulting key will be fixed*, and the 128-bit key the parties derive will have only about 75 bits of entropy (i.e., there are only 2^{75} or so possibilities for the key).

What we need is a generic solution for deriving a key from a high-entropy (but not necessarily uniform) shared secret. Before continuing, we define the notion of entropy we consider here.

DEFINITION 5.12 *A probability distribution $\mathcal{X}$ has m* bits of min-entropy *if for every fixed value x it holds that $\Pr_{X \leftarrow \mathcal{X}}[X = x] \leq 2^{-m}$. That is, even the most likely outcome occurs with probability at most 2^{-m}.*

The uniform distribution over a set of size S has min-entropy $\log S$. A distribution in which one element occurs with probability 1/10 and 90 elements each occur with probability 1/100 has min-entropy $\log 10 \approx 3.3$. The min-entropy of a distribution measures the probability with which an attacker can guess a value sampled from that distribution; the attacker's best strategy is to guess the most likely value, and so if the distribution has min-entropy m the attacker guesses correctly with probability at most 2^{-m}. This explains why min-entropy (rather than other notions of entropy) is useful in our context. An extension of min-entropy, called *computational min-entropy*, is defined as above except that the distribution is only required to be *computationally indistinguishable* from a distribution with the given min-entropy. (The notion of computational indistinguishability is formally defined in Section 7.8.)

A *key-derivation function* provides a way to obtain a uniformly distributed string from any distribution with high (computational) min-entropy. It is not hard to see that if we model a hash function H as a random oracle, then H serves as a good key-derivation function. Consider an attacker's uncertainty about $H(X)$, where X is sampled from a distribution with min-entropy m (as a technical point, we require the distribution to be independent of H). Each of the attacker's queries to H can be viewed as a "guess" for the value of X; by assumption on the min-entropy of the distribution, an attacker making q queries to H will query $H(X)$ with probability at most $q \cdot 2^{-m}$. If the attacker does not query X to H, then $H(X)$ is a uniform string.

It is also possible to design key-derivation functions, without relying on the random-oracle model, using keyed hash functions called *(strong) extractors*. The key for the extractor must be uniform, but need not be kept secret. One standard for this is called HKDF; see the references at the end of the chapter.

5.6.5 Commitment Schemes

A *commitment scheme* allows one party to "commit" to a message m by sending a commitment value com, while obtaining the following seemingly contradictory properties:

- *Hiding:* the commitment reveals nothing about m.
- *Binding:* it is infeasible for the committer to output a commitment com that it can later "open" as two different messages m, m'. (In this sense, com truly "commits" the committer to some well-defined value.)

A commitment scheme can be seen as a digital envelope: sealing a message in an envelope and handing it over to another party provides privacy (until the

envelope is opened) and binding (since the envelope is sealed).

Formally, a (non-interactive) commitment scheme is defined by a randomized algorithm Gen that outputs public parameters params and an algorithm Com that takes params and a message $m \in \{0,1\}^n$ and outputs a commitment com; we will make the randomness used by Com explicit, and denote it by r. A sender commits to m by choosing uniform r, computing $\mathsf{com} := \mathsf{Com}(\mathsf{params}, m; r)$, and sending it to a receiver. The sender can later decommit com and reveal m by sending m, r to the receiver; the receiver verifies this by checking that $\mathsf{Com}(\mathsf{params}, m; r) \stackrel{?}{=} \mathsf{com}$.

Hiding, informally, means that com reveals nothing about m; binding means that it is impossible to output a commitment com that can be opened two different ways. We define these properties formally now.

The commitment hiding experiment $\mathsf{Hiding}_{\mathcal{A},\mathsf{Com}}(n)$**:**

1. *Parameters* $\mathsf{params} \leftarrow \mathsf{Gen}(1^n)$ *are generated.*
2. *The adversary* $\mathcal{A}$ *is given input* params*, and outputs a pair of messages* $m_0, m_1 \in \{0,1\}^n$*.*
3. *A uniform* $b \in \{0,1\}$ *is chosen and* $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{params}, m_b; r)$ *is computed.*
4. *The adversary* $\mathcal{A}$ *is given* com *and outputs a bit* b'*.*
5. *The output of the experiment is* 1 *if and only if* $b' = b$*.*

The commitment binding experiment $\mathsf{Binding}_{\mathcal{A},\mathsf{Com}}(n)$**:**

1. *Parameters* $\mathsf{params} \leftarrow \mathsf{Gen}(1^n)$ *are generated.*
2. $\mathcal{A}$ *is given input* params *and outputs* $(\mathsf{com}, m, r, m', r')$*.*
3. *The output of the experiment is defined to be* 1 *if and only if* $m \neq m'$ *and* $\mathsf{Com}(\mathsf{params}, m; r) = \mathsf{com} = \mathsf{Com}(\mathsf{params}, m'; r')$*.*

DEFINITION 5.13 *A commitment scheme* Com *is* secure *if for all* PPT *adversaries* $\mathcal{A}$ *there is a negligible function* negl *such that*

$$\Pr\left[\mathsf{Hiding}_{\mathcal{A},\mathsf{Com}}(n) = 1\right] \leq \tfrac{1}{2} + \mathsf{negl}(n)$$

and

$$\Pr\left[\mathsf{Binding}_{\mathcal{A},\mathsf{Com}}(n) = 1\right] \leq \mathsf{negl}(n).$$

It is easy to construct a secure commitment scheme from a random oracle H. To commit to a message m, the sender chooses uniform $r \in \{0,1\}^n$ and outputs $\mathsf{com} := H(m\|r)$. (In the random-oracle model, Gen and params are not needed since H, in effect, serves as the public parameters of the scheme.) Intuitively, hiding follows from the fact that an adversary queries $H(\star\|r)$ with

only negligible probability (since r is a uniform n-bit string); if it never makes a query of this form then $H(m\|r)$ reveals nothing about m. Binding follows from the fact that H is collision resistant.

Commitment schemes can be constructed without random oracles (in fact, from one-way functions), but the details are beyond the scope of this book.

References and Additional Reading

Collision-resistant hash functions were formally defined by Damgård [52]. Additional discussion regarding notions of security for hash functions besides collision resistance can be found in [120, 150]. The Merkle–Damgård transform was introduced independently by Damgård and Merkle [53, 123]

HMAC was introduced by Bellare et al. [14] and later standardized [131].

The small-space birthday attack described in Section 5.4.2 relies on a cycle-finding algorithm of Floyd. Related algorithms and results are described at `http://en.wikipedia.org/wiki/Cycle_detection`. The idea for finding meaningful collisions using the small-space attack is by Yuval [180]. The possibility of parallelizing collision-finding attacks, which can offer significant speedups in practice, is discussed in [170]. Time/space tradeoffs for function inversion were introduced by Hellman [87], with practical improvements—not discussed here—given by Rivest (unpublished) and by Oechslin [134].

The first formal treatment of the random-oracle model was given by Bellare and Rogaway [21], although the idea of using a "random-looking" function in cryptographic applications had been suggested previously, most notably by Fiat and Shamir [65]. Proper instantiation of a random oracle based on concrete cryptographic hash functions is discussed in [21, 22, 23, 48]. The seminal negative result concerning the random-oracle model is that of Canetti et al. [41], who show (contrived) schemes that are secure in the random-oracle model but are insecure for *any* concrete instantiation of the random oracle.

Merkle trees were introduced in [121]. Key-derivation functions used in practice include HKDF, PBKDF2, and bcrypt. See [109] for a formal treatment of the problem and an analysis of HKDF.

Exercises

5.1 Provide formal definitions for second preimage resistance and preimage resistance. 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.

5.2 Let (Gen_1, H_1) and (Gen_2, H_2) be two hash functions. Define (Gen, H) so that Gen runs Gen_1 and Gen_2 to obtain keys s_1 and s_2, respectively. Then define $H^{s_1,s_2}(x) = H_1^{s_1}(x) \| H_2^{s_2}(x)$.

(a) Prove that if at least one of (Gen_1, H_1) and (Gen_2, H_2) is collision resistant, then (Gen, H) is collision resistant.

(b) Determine whether an analogous claim holds for second preimage resistance and preimage resistance, respectively. Prove your answer in each case.

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

5.4 Provide a formal proof of Theorem 5.4 (i.e., describe the reduction).

5.5 Generalize the Merkle–Damgård transform (Construction 5.3) for the case when the fixed-length hash function h has input length $n + \kappa$ (with $\kappa > 0$) and output length n, and the length of the input to H should be encoded as an ℓ-bit value (as discussed in Section 5.3.2). Prove collision resistance of (Gen, H), assuming collision resistance of (Gen, h).

5.6 For each of the following modifications to the Merkle–Damgård transform (Construction 5.3), determine whether the result is collision resistant. If yes, provide a proof; if not, demonstrate an attack.

(a) Modify the construction so that the input length is not included at all (i.e., output z_B and not $z_{B+1} = h^s(z_B \| L)$). (Assume the resulting hash function is only defined for inputs whose length is an integer multiple of the block length.)

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

(c) Instead of using an IV, just start the computation from x_1. That is, define $z_1 := x_1$ and then compute $z_i := h^s(z_{i-1} \| x_i)$ for $i = 2, \ldots, B+1$ and output z_{B+1} as before.

(d) Instead of using a fixed IV, set $z_0 := L$ and then compute $z_i := h^s(z_{i-1} \| x_i)$ for $i = 1, \ldots, B$ and output z_B.

5.7 Assume collision-resistant hash functions exist. Show a construction of a fixed-length hash function (Gen, h) that is *not* collision resistant, but such that the hash function (Gen, H) obtained from the Merkle–Damgård transform to (Gen, h) as in Construction 5.3 *is* collision resistant.

5.8 Prove or disprove: if (Gen, h) is preimage resistant, then so is the hash function (Gen, H) obtained by applying the Merkle–Damgård transform to (Gen, h) as in Construction 5.3.

5.9 Prove or disprove: if (Gen, h) is second preimage resistant, then so is the hash function (Gen, H) obtained by applying the Merkle–Damgård transform to (Gen, h) as in Construction 5.3.

5.10 Before HMAC, it was common to define a MAC for arbitrary-length messages by $\mathsf{Mac}_{s,k}(m) = H^s(k\|m)$ where H is a collision-resistant hash function.

(a) Show that this is never a secure MAC when H is constructed via the Merkle–Damgård transform. (Assume the hash key s is known to the attacker, and only k is kept secret.)

(b) Prove that this is a secure MAC if H is modeled as a random oracle.

5.11 Prove that the construction of a pseudorandom function given in Section 5.5.1 is secure in the random-oracle model.

5.12 Prove Theorem 5.11.

5.13 Show how to find a collision in the Merkle tree construction if t is not fixed. Specifically, show how to find two sets of inputs $x_1, \ldots, x_t$ and $x'_1, \ldots, x'_{2t}$ such that $\mathcal{MT}_t(x_1, \ldots, x_t) = \mathcal{MT}_{2t}(x'_1, \ldots, x'_{2t})$.

5.14 Consider the scenario introduced in Section 5.6.2 in which a client stores files on a server and wants to verify that files are returned unmodified.

(a) Provide a formal definition of security for this setting.

(b) Formalize the construction based on Merkle trees as discussed in Section 5.6.2.

(c) Prove that your construction is secure relative to your definition under the assumption that (Gen_H, H) is collision resistant.

5.15 Prove that the commitment scheme discussed in Section 5.6.5 is secure in the random-oracle model.

Chapter 6

Practical Constructions of Symmetric-Key Primitives

In previous chapters we have demonstrated how secure encryption schemes and message authentication codes can be constructed from cryptographic primitives such as pseudorandom generators, pseudorandom permutations, and hash functions. One question we have not yet addressed, however, is how these cryptographic primitives are constructed in the first place, or even whether they exist at all! In the next chapter we will study this question from a theoretical vantage point, and show constructions of pseudorandom generators and pseudorandom permutations based on quite weak assumptions. (It turns out that hash functions are more difficult to construct, and appear to require stronger assumptions. We will see a provably secure construction of hash functions in Section 8.4.2.) In this chapter, our focus will be on comparatively heuristic, but far more efficient, constructions of these primitives that are widely used in practice.

As just mentioned, the constructions that we will explore in this chapter are heuristic in the sense that they cannot be proven secure based on any weaker assumption. These constructions are, however, based on a number of design principles, some of which can be justified by theoretical analysis. Perhaps more importantly, many of these constructions have withstood years of public scrutiny and attempted cryptanalysis and, given this, it is quite reasonable to assume the security of these constructions.

In some sense there is no fundamental difference between assuming, say, that factoring is hard and assuming that AES (a block cipher we will study in detail later in this chapter) is a pseudorandom permutation. There is, however, a significant *qualitative* difference between these assumptions.[1] The primary difference is that the former assumption is more believable since it seemingly relates to a weaker requirement: the assumption that large integers are hard to factor is arguably more natural than the assumption that AES with a uniform key is indistinguishable from a random permutation. Other relevant differences between the assumptions are that factoring has been studied much longer than the problem of distinguishing AES from a random permutation,

[1] It should be clear that the discussion in this paragraph is informal, as we cannot formally argue about any of this when we cannot even prove that factoring is hard in the first place!

and was recognized as a hard problem well before the advent of cryptographic schemes based on it.

To summarize, it is reasonable to assume that the recommended constructions described in this chapter are secure, and people are comfortable relying on such assumptions in practice. Still, it would be preferable to base security of cryptographic primitives on weaker and more long-standing assumptions. As we will see in Chapter 7, this is (in principle) possible; unfortunately, the constructions we will see there are orders of magnitude less efficient than the constructions described here, and as such are not useful in practice.

The Aim of This Chapter

The main aims of this chapter are (1) to present some design principles used in the construction of modern cryptographic primitives, and (2) to introduce the reader to some popular constructions used extensively in the real world. We caution that:

- It is *not* the aim of this chapter to teach readers how to design new cryptographic primitives. On the contrary, we believe that the design of new primitives requires significant expertise and effort, and is not something to be attempted lightly. Those who are interested in developing additional expertise in this area are advised to read the more advanced references included at the end of the chapter.

- It is *not* our intent to present all the low-level details of the various primitives we discuss here, and our descriptions should not be relied upon for implementation. In fact, our descriptions are sometimes purposefully inaccurate, as we omit certain details that are not relevant to the broader conceptual point we are trying to emphasize.

6.1 Stream Ciphers

Recall from Section 3.3.1 that a stream cipher is defined by two deterministic algorithms $(\mathsf{Init}, \mathsf{GetBits})$. The Init algorithm takes as input a key k and an (optional) initialization vector IV and returns some initial state st. The $\mathsf{GetBits}$ algorithm can be used to generate an infinite stream of bits $y_1, y_2, \ldots$ based on st. The main requirement of a stream cipher is that it should behave like a pseudorandom generator, namely, when k is chosen uniformly at random the resulting sequence $y_1, y_2, \ldots$ should be indistinguishable from a sequence of uniform and independent bits by any computationally bounded attacker. (In Section 3.6.1 we noted that stream ciphers must sometimes satisfy stronger security requirements. We do not explicitly address this here.)

We have already pointed out (see the end of Section 3.5.1) that stream ciphers can be constructed easily from block ciphers, which are a stronger primitive. The primary motivation for using the dedicated stream-cipher constructions introduced in this section is efficiency, especially in resource-constrained environments (e.g., in hardware where there may be a desire to keep the number of gates small). Attacks have been shown against several recent constructions of stream ciphers, however, and their security appears much more tenuous than is the case for block ciphers. We therefore recommend using block ciphers (possibly in stream-cipher mode) when possible.

6.1.1 Linear-Feedback Shift Registers

We begin by discussing *linear-feedback shift registers* (LFSRs). These have been used historically for pseudorandom-number generation, as they are extremely efficient to implement in hardware, and generate output having good statistical properties. By themselves, however, they do *not* give cryptographically strong pseudorandom generators, and in fact we will show an easy key-recovery attack on LFSRs. Nevertheless, LFSRs can be used as a component in building stream ciphers with better security.

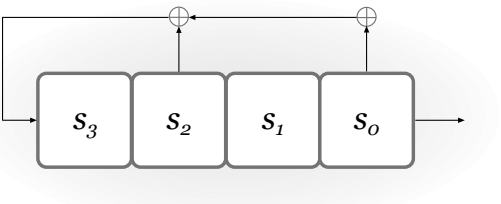


FIGURE 6.1: A linear-feedback shift register.

An LFSR consists of an array of n *registers* $s_{n-1}, \ldots, s_0$ along with a feedback loop specified by a set of n *feedback coefficients* $c_{n-1}, \ldots, c_0$. (See Figure 6.1.) The size of the array is called the *degree* of the LFSR. Each register stores a single bit, and the state st of the LFSR at any point in time is simply the set of bits contained in the registers. The state of the LFSR is updated in each of a series of "clock ticks" by shifting the values in all the registers to the right, and setting the new value of the left-most register equal to the XOR of some subset of the current registers, with the subset determined by the feedback coefficients. That is, if the state at some time t is $s_{n-1}^{(t)}, \ldots, s_0^{(t)}$, then the state after the next clock tick is $s_{n-1}^{(t+1)}, \ldots, s_0^{(t+1)}$ with

$$\begin{aligned} s_i^{(t+1)} &:= s_{i+1}^{(t)}, \qquad\qquad i = 0, \ldots, n-2 \\ s_{n-1}^{(t+1)} &:= \bigoplus_{i=0}^{n-1} c_i \, s_i^{(t)}. \end{aligned}$$

Figure 6.1 shows a degree-4 LFSR with $c_0 = c_2 = 1$ and $c_1 = c_3 = 0$.

At each clock tick, the LFSR outputs the value of the right-most register s_0. If the initial state of the LFSR is $s_{n-1}^{(0)}, \ldots, s_0^{(0)}$, the first n bits of the output stream are exactly $s_0^{(0)}, \ldots, s_{n-1}^{(0)}$. The next output bit is $s_{n-1}^{(1)} = \bigoplus_{i=0}^{n-1} c_i \, s_i^{(0)}$. If we denote the output bits by $y_1, y_2, \ldots$, where $y_i = s_0^{(i-1)}$, then

$$\begin{aligned} y_i &= s_{i-1}^{(0)}, && i = 1, \ldots, n \\ y_i &= \bigoplus_{j=0}^{n-1} c_j \, y_{i-n+j-1} && i > n. \end{aligned}$$

As an example using the LFSR from Figure 6.1, if the initial state is $(0, 0, 1, 1)$ then the states for the first few time periods are

$$\begin{gathered} (0, 0, 1, 1) \\ (1, 0, 0, 1) \\ (1, 1, 0, 0) \\ (1, 1, 1, 0) \\ (1, 1, 1, 1) \end{gathered}$$

and the output (which can be read off the right-most column of the above) is the stream of bits $1, 1, 0, 0, 1, \ldots$.

The state of the LFSR consists of n bits; thus, the LFSR can cycle through at most 2^n possible states before repeating. When the states repeat the output bits repeat, and this means that the output sequence will begin repeating after at most 2^n output bits have been generated. A *maximum-length* LFSR cycles through all $2^n - 1$ nonzero states before repeating. (Note that if the all-0 state is ever realized then the LFSR remains in that state forever, which is why we exclude it.) Whether an LFSR is maximal length or not depends only on the feedback coefficients; if it is maximal length then, once it is initialized in any nonzero state, it will cycle through all $2^n - 1$ nonzero states. It is well understood how to set the feedback coefficients to obtain a maximal-length LFSR, although the details are beyond the scope of this book.

Reconstruction attacks. The output of a maximal-length LFSR of degree n has good statistical properties; for example, every n-bit string occurs with roughly equal frequency in the output stream of the LFSR. Nevertheless, LFSRs are not good pseudorandom generators for cryptographic purposes because their output is predictable. This follows from the fact that an attacker can reconstruct the entire state of a degree-n LFSR after observing at most $2n$ output bits. To see this, assume both the initial state and the feedback coefficients of some LFSR are unknown. The first n output bits $y_1, \ldots, y_n$ of the LFSR exactly reveal the initial state. Given the next n output bits $y_{n+1}, \ldots, y_{2n}$, the attacker can set up a system of n linear equations in the n

unknowns $c_0, \ldots, c_{n-1}$:

$$\begin{aligned} y_{n+1} &= c_{n-1}\, y_n \oplus \cdots \oplus c_0\, y_1 \\ &\vdots \\ y_{2n} &= c_{n-1}\, y_{2n-1} \oplus \cdots \oplus c_0\, y_n. \end{aligned}$$

One can show that the above equations are linearly independent (modulo 2) for a maximal-length LFSR, and so uniquely determine the feedback coefficients. (The solution can be found efficiently using linear algebra.) With the feedback coefficients known, all subsequent output bits of the LFSR can be easily computed.

6.1.2 Adding Nonlinearity

The linear relationships between the outputs bits of an LFSR are exactly what enable an easy attack. To thwart such attacks, we must introduce some nonlinearity, i.e., some operations other than XORs. There are several different approaches to doing so, and we only explore some of them here.

Nonlinear feedback. One obvious way to modify LFSRs is to make the feedback loop nonlinear. A nonlinear-feedback shift register (FSR) will again consist of an array of registers, each containing a single bit. As before, the state of the FSR is updated in each of a series of clock ticks by shifting the values in all the registers to the right; now, however, the new value of the left-most register is a *nonlinear* function of the current registers. In other words, if the state at some time t is $s_0^{(t)}, \ldots, s_{n-1}^{(t)}$ then the state after the next clock tick is $s_0^{(t+1)}, \ldots, s_{n-1}^{(t+1)}$ with

$$\begin{aligned} s_i^{(t+1)} &:= s_{i+1}^{(t)}, \qquad i = 0, \ldots, n-2 \\ s_{n-1}^{(t+1)} &:= g(s_0^{(t)}, \ldots, s_{n-1}^{(t)}) \end{aligned}$$

for some nonlinear function g. As before, the FSR outputs the value of the right-most register s_0 at each clock tick.

It is possible to design nonlinear FSRs with maximal length and such that the output has good statistical properties.

Nonlinear combination generators. Another approach is to introduce nonlinearity in the output sequence. In the most basic case, we could have an LFSR as before (where the new value of the left-most register is again computed as a linear function of the current registers), but where the output at each clock tick is a nonlinear function g of all the current registers, rather than just the right-most register. It is important here that g be *balanced* in the sense that $\Pr[g(s_0, \ldots, s_{n-1}) = 1] \approx 1/2$ (where the probability is over uniform choice of $s_0, \ldots, s_{n-1}$); otherwise, although it might be difficult to

reconstruct the entire state of the LFSR based on the output, the output stream will be biased and hence easily distinguishable from uniform.

A variant of the above is to use several LFSRs (with each individual output stream computed, as before, by simply taking the value of the right-most register of each LFSR), and to generate the actual output stream by combining the output of the individual LFSRs in some nonlinear way. This yields what is known as a (nonlinear) *combination generator*. The individual LFSRs need not have the same degree, and in fact the cycle length of the combination generator will be maximized if they do *not* have the same degree. Here, care must be taken to ensure that the output stream of the combination generator is not too highly correlated with any of the output streams of the individual LFSRs; high correlation can lead to attacks on the individual LFSRs, thereby defeating the purpose of using several LFSRs in the construction.

6.1.3 Trivium

To illustrate the ideas from the previous section, we briefly describe the stream cipher *Trivium*. This stream cipher was selected as part of the portfolio of the eSTREAM project, a European effort completed in 2008 whose goal was to identify new stream ciphers. Trivium was designed to have a simple description and to admit a compact hardware implementation.

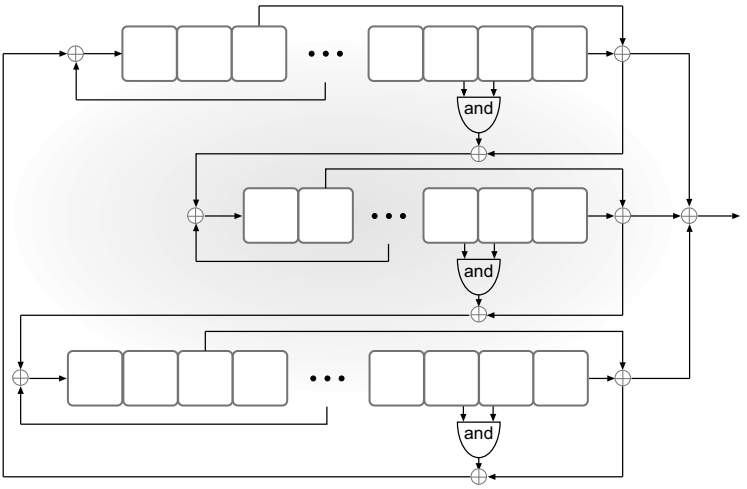


FIGURE 6.2: A schematic illustration of Trivium with (from top to bottom) three coupled, nonlinear FSRs A, B, and C.

Trivium uses three coupled, nonlinear FSRs denoted by A, B, and C and having degree 93, 84, and 111, respectively. (See Figure 6.2.) The state st of Trivium is simply the 288 bits comprising the values in all the registers

of these FSRs. The GetBits algorithm for Trivium works as follows: At each clock tick, the output of each FSR is the XOR of its right-most register and one additional register; the output of Trivium is the XOR of the output bits of the three FSRs. The FSRs are *coupled*: at each clock tick, the new value of the left-most register of each FSR is computed as a function of one of the registers in the same FSR and a subset of the registers from a second FSR. (The feedback function for A depends on one register of A and four registers of C; the feedback function for B depends on one register of B and four registers of A; and the feedback function for C depends on one register of C and four registers of B.) The feedback function in each case is nonlinear.

The Init algorithm of Trivium accepts an 80-bit key and an 80-bit IV. The key is loaded into the 80 left-most registers of A, and the IV is loaded into the 80 left-most registers of B. The remaining registers are set to 0, except for the three right-most registers of C, which are set to 1. Then GetBits is run $4 \cdot 288$ times (with output discarded), and the resulting state is taken as st_0.

To date, no cryptanalytic attacks better than exhaustive key search are known against the full Trivium cipher.

6.1.4 RC4

LFSRs are efficient when implemented in hardware, but have poor performance in software. For this reason, alternate designs of stream ciphers have been explored. A prominent example is RC4, which was designed by Ron Rivest in 1987. RC4 is remarkable for its speed and simplicity, and resisted serious attack for several years. It is widely used today, and we discuss it for this reason; we caution the reader, however, that recent attacks have shown serious cryptographic weaknesses in RC4 and it should no longer be used.

ALGORITHM 6.1
Init algorithm for RC4

Input: 16-byte key k
Output: Initial state (S, i, j)
(Note: All addition is done modulo 256)

for $i = 0$ to 255:
  $S[i] := i$
  $k[i] := k[i \bmod 16]$
$j := 0$
for $i = 0$ to 255:
  $j := j + S[i] + k[i]$
  Swap $S[i]$ and $S[j]$
$i := 0$, $j := 0$
return (S, i, j)

The state of RC4 is a 256-byte array S, which always contains a permutation of the elements $0, \ldots, 255$, along with two values $i, j \in \{0, \ldots, 255\}$. The Init

algorithm for RC4 is presented as Algorithm 6.1. For simplicity we assume a 16-byte (128-bit) key k, although the algorithm can handle keys between 1 byte and 256 bytes long. We index the bytes of S as $S[0], \ldots, S[255]$, and the bytes of the key as $k[0], \ldots, k[15]$.

During initialization, S is first set to the identity permutation (i.e., with $S[i] = i$ for all i) and k is expanded to 256 bytes by repetition. Then each entry of S is swapped at least once with another entry of S in a "pseudorandom" location. The indices i, j are set to 0, and (S, i, j) is output as the initial state.

The state is then used to generate a sequence of output bits, as shown in Algorithm 6.2. The index i is simply incremented (modulo 256), and j is changed in some "pseudorandom" way. Entries $S[i]$ and $S[j]$ are swapped, and the value of S at position $S[i] + S[j]$ (again computed modulo 256) is output. Note that each entry of S is swapped with some other entry of S (possibly itself) at least once every 256 iterations, ensuring good "mixing" of the permutation S.

ALGORITHM 6.2
GetBits algorithm for RC4

Input: Current state (S, i, j)
Output: Output byte y; updated state (S, i, j)
(Note: All addition is done modulo 256)

$i := i + 1$
$j := j + S[i]$
Swap $S[i]$ and $S[j]$
$t := S[i] + S[j]$
$y := S[t]$
return $(S, i, j), y$

RC4 was not designed to take an IV as input; however, in practice an IV is often incorporated by simply concatenating it with the actual key k' before initialization. That is, a random IV of the desired length is chosen, k is set to be equal to the concatenation of IV and k' (this can be done by either prepending or appending IV), and then Init is run as in Algorithm 6.1 to generate an initial state. Output bits are then produced using Algorithm 6.2 exactly as before. Assuming RC4 is being used in unsynchronized mode (see Section 3.6.1), the IV would then be sent in the clear to the receiver—who presumably already has the actual key k'—thus enabling them to generate the same initial state and hence the same output stream. This method of incorporating an IV is used in the *Wired Equivalent Privacy* (WEP) encryption standard for protecting communications in 802.11 wireless networks.

One should be concerned by this relatively ad hoc way of modifying RC4 to accept an IV. Even if RC4 were a secure stream cipher when using (only) a key as originally designed, there is no reason to believe that it should be

secure when modified to use an IV in this way. Indeed, contrary to the key, the IV is revealed to an attacker (since it is sent in the clear); furthermore, using different IVs with the same fixed key k'—as would be done when using RC4 in unsynchronized mode—means that *related* values k are being used to initialize the state of RC4. As we will see below, both of these issues lead to attacks when RC4 is used in this fashion.

Attacks on RC4. Although RC4 is pervasive in modern systems, various attacks on RC4 have been known for several years. Due to this, RC4 should no longer be used; instead, a more modern stream cipher or block cipher should be used in its place.

We begin by demonstrating a simple statistical attack on RC4 that does not rely on the honest parties' using an IV. The attack exploits the fact that the second output byte of RC4 is (slightly) *biased* toward 0. Let S_t denote the state of the array S after t iterations of GetBits, with S_0 denoting the initial state. Treating S_0 (heuristically) as a uniform permutation of $\{0, \ldots, 255\}$, with probability $1/256 \cdot (1 - 1/255) \approx 1/256$, it holds that $S_0[2] = 0$ and $X \stackrel{\text{def}}{=} S_0[1] \neq 2$. Assume for a moment that this is the case. In the first iteration of GetBits, the value of i is incremented to 1, and j is set equal to $S_0[i] = S_0[1] = X$. Then $S_0[1]$ and $S_0[X]$ are swapped, so that at the end of the iteration we have $S_1[X] = S_0[1] = X$. In the second iteration, i is incremented to 2 and j is assigned the value

$$j + S_1[i] = X + S_1[2] = X + S_0[2] = X,$$

since $S_0[2] = 0$. Then $S_1[2]$ and $S_1[X]$ are swapped, so that $S_2[X] = S_1[2] = S_0[2] = 0$ and $S_2[2] = S_1[X] = X$. Finally, the value of S_2 at position $S_2[i]+S_2[j] = S_2[2]+S_2[X] = X$ is output; this is exactly the value $S_2[X] = 0$.

When $S_0[2] \neq 0$ the second output byte is uniformly distributed. Overall, then, the probability that the second output byte is 0 is

$$\begin{aligned}\Pr[S_0[2] = 0 \text{ and } S_0[1] \neq 2] + \frac{1}{256} \cdot \Big(1 - \Pr[S_0[2] = 0 \text{ and } S_0[1] \neq 2]\Big) \\ = \frac{1}{256} + \frac{1}{256} \cdot \left(1 - \frac{1}{256}\right) \approx \frac{2}{256},\end{aligned}$$

or twice what would be expected for a uniform value.

By itself the above might not be viewed as a particularly serious attack, although it does seem to indicate underlying structural problems with RC4. A more serious attack against RC4 is possible when an IV is incorporated by prepending it to the key. This attack can used to recover the key, regardless of its length, and is thus more serious than a distinguishing attack such as the one described above. Importantly, this attack can be used to completely break the WEP encryption standard mentioned earlier, and was influential in getting the standard replaced.

The core of the attack is a way to extend knowledge of the first n bytes of k to knowledge of the first $(n + 1)$ bytes of k. Note that when an IV is

prepended to the actual key k' (so $k = IV \| k'$), the first few bytes of k are given to the attacker for free! If the IV is n bytes long, then the adversary can use this attack to first recover the $(n+1)$st byte of k (which is the first byte of the real key k'), then the next byte of k, and so on, until it deduces the entire key.

Assume the IV is 3 bytes long, as is the case for WEP. The attacker waits until the first two bytes of the IV have a specific form. The attack can be carried out with several possibilities for the first two bytes of the IV, but we look at the case where the IV takes the form $IV = (3, 255, X)$ for X an arbitrary byte. This means, of course, that $k[0] = 3, k[1] = 255$, and $k[2] = X$ in Algorithm 6.1. One can check that after the first four iterations of the second loop of Init, we have

$$S[0] = 3, \quad S[1] = 0, \quad S[3] = X + 6 + k[3]. \tag{6.1}$$

In the next 252 iterations of the Init algorithm, i is always greater than 3. So the values of $S[0], S[1]$, and $S[3]$ are not subsequently modified as long as j never takes on the values 0, 1, or 3. If we (heuristically) treat j as taking on a uniform value in each iteration, this means that $S[0], S[1]$, and $S[3]$ are not subsequently modified with probability $(253/256)^{252} \approx 0.05$, or 5% of the time. Assuming this is the case, the first byte output by GetBits will be $S[3] = X + 6 + k[3]$; since X is known, this reveals $k[3]$.

So, the attacker knows that 5% of the time the first byte of the output is correlated with $k[3]$ as described above. (This is much better than random guessing, which is correct $1/256 = 0.4\%$ of the time.) Thus, by collecting sufficiently many samples of the first byte of the output—for several IVs having the correct form—the attacker gets a high-confidence estimate for $k[3]$.

6.2 Block Ciphers

Recall from Section 3.5.1 that a block cipher is an efficient, keyed permutation $F : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^\ell$. This means the function F_k defined by $F_k(x) \stackrel{\text{def}}{=} F(k, x)$ is a bijection (i.e., a permutation), and moreover F_k and its inverse F_k^{-1} are efficiently computable given k. We refer to n as the *key length* and ℓ as the *block length* of F, and here we explicitly allow them to differ. The key length and block length are now fixed constants, whereas in Chapter 3 they were viewed as functions of a security parameter. This puts us in the setting of concrete security rather than asymptotic security.[2] The concrete

[2] Although a block cipher with fixed key length has no "security parameter" to speak of, we still view security as depending on the length of the key and thus denote this value by n.

security requirements for block ciphers are quite stringent, and a block cipher is generally only considered "good" if the best known attack (without preprocessing) has time complexity roughly equivalent to a brute-force search for the key. Thus, if a cipher with key length $n = 256$ can be broken in time 2^{128}, the cipher is (generally) considered insecure even though a 2^{128}-time attack is still infeasible. In contrast, in an asymptotic setting an attack of complexity $2^{n/2}$ is not considered efficient since it requires exponential time (and thus a cipher where such an attack is possible might still satisfy the definition of being a pseudorandom permutation). In the concrete setting, however, we must worry about the actual complexity of the attack (rather than its asymptotic behavior). Furthermore, there is a concern that existence of such an attack may indicate some more fundamental weakness in the design of the cipher.

Block ciphers are designed to behave, at a minimum, as (strong) pseudorandom permutations; see Definition 3.28. Modeling block ciphers as pseudorandom permutations allows proofs of security for constructions based on block ciphers, and also makes explicit the necessary requirements of a block cipher. A solid understanding of what block ciphers are supposed to achieve is instrumental in their design. The view that block ciphers should be modeled as pseudorandom permutations has, at least in the recent past, served as a major influence in their design. As an example, the call for proposals for the recent Advanced Encryption Standard (AES) that we will encounter later in this chapter stated the following evaluation criterion:

> *The security provided by an algorithm is the most important factor.... Algorithms will be judged on the following factors...*
>
> - *The extent to which the algorithm output is indistinguishable from a random permutation ...*

Modern block ciphers are suitable for all the constructions using pseudorandom permutations (or pseudorandom functions) we have seen in this book.

Often, block ciphers are designed (and assumed) to satisfy even stronger security properties, as we discuss briefly in Section 6.3.1.

Notwithstanding the fact that block ciphers are not, on their own, encryption schemes, the standard terminology for attacks on a block cipher F is:

- In a *known-plaintext attack*, the attacker is given pairs of inputs/outputs $\{(x_i, F_k(x_i))\}$ (for an unknown key k), with the $\{x_i\}$ outside the attacker's control.
- In a *chosen-plaintext attack*, the attacker is given $\{F_k(x_i)\}$ (again, for an unknown key k) for a series of inputs $\{x_i\}$ chosen by the attacker.
- In a *chosen-ciphertext attack*, the attacker is given $\{F_k(x_i)\}$ for $\{x_i\}$ chosen by the attacker, as well as $\{F_k^{-1}(y_i)\}$ for chosen $\{y_i\}$.

Viewing the key length as a parameter makes sense when comparing block ciphers with different key lengths, or when using a block cipher that supports keys of different lengths.

Besides using the above to distinguish F_k from a uniform permutation, we will also be interested in *key-recovery attacks* in which the attacker is able to recover the key k after interacting with F_k. (This is stronger than being able to distinguish F_k from uniform.)

With respect to this taxonomy, a pseudorandom permutation cannot be distinguished from a uniform permutation under a chosen-plaintext attack, while a strong pseudorandom permutation cannot be so distinguished even under a chosen-ciphertext attack.

6.2.1 Substitution-Permutation Networks

A block cipher must behave like a random permutation. There are $2^\ell!$ permutations on ℓ-bit strings, so representing an arbitrary permutation with an ℓ-bit block length requires $\log(2^\ell!) \approx \ell \cdot 2^\ell$ bits. This is impractical for $\ell > 20$ and infeasible for $\ell > 50$. (Looking ahead, modern block ciphers have block lengths $\ell \geq 128$.) The challenge when designing a block cipher is to construct a set of permutations with a *concise* description (namely, a short key) that behaves like a random permutation. In particular, just as evaluating a random permutation at two inputs that differ in only a single bit should yield two (almost) independent outputs (they are not completely independent since they cannot be equal), so too changing one bit of the input to $F_k(\cdot)$, where k is uniform and unknown to an attacker, should yield an (almost) independent result. This implies that a one-bit change in the input should "affect" every bit of the output. (Note that this does not mean that all the output bits will be changed—that would be different behavior than one would expect for a random permutation. Rather, we just mean informally that each bit of the output is changed with probability roughly half.) This takes some work to achieve.

The confusion-diffusion paradigm. In addition to his work on perfect secrecy, Shannon also introduced a basic paradigm for constructing concise, random-looking permutations. The basic idea is to construct a random-looking permutation F with a large block length from many smaller random (or random-looking) permutations $\{f_i\}$ with small block length. Let us see how this works on the most basic level. Say we want F to have a block length of 128 bits. We can define F as follows: the key k for F will specify 16 permutations $f_1, \ldots, f_{16}$ that each have an 8-bit (1-byte) block length.[3] Given an input $x \in \{0,1\}^{128}$, we parse it as 16 bytes $x_1 \cdots x_{16}$ and then set

$$F_k(x) = f_1(x_1) \| \cdots \| f_{16}(x_{16}). \tag{6.2}$$

These round functions $\{f_i\}$ are said to introduce *confusion* into F.

[3] An arbitrary permutation on 8 bits can be represented using $\log(2^8!) \approx 1600$ bits, so the length of the key for F is about $16 \cdot 1600$ bits, or 3 kbytes. This is much smaller than the $\approx 128 \cdot 2^{128}$ bits that would be required to specify an arbitrary permutation on 128 bits.

It should be immediately clear, however, that F as defined above will *not* be pseudorandom. Specifically, if x and x' differ only in their first bit then $F_k(x)$ and $F_k(x')$ will differ only in their first byte (regardless of the key k). In contrast, if F were a truly random permutation then changing the first bit of the input would be expected to affect all bytes of the output.

For this reason, a *diffusion* step is introduced whereby the bits of the output are permuted, or "mixed," using a *mixing permutation*. This has the effect of spreading a local change (e.g., a change in the first byte) throughout the entire block. The confusion/diffusion steps—together called a *round*—are repeated multiple times. This helps ensure that changing a single bit of the input will affect all the bits of the output.

As an example, a two-round block cipher following this approach would operate as follows. First, confusion is introduced by computing the intermediate result $f_1(x_1)\| \cdots \|f_{16}(x_{16})$ as in Equation (6.2). The bits of the result are then "shuffled," or re-ordered, to give x'. Then $f'_1(x'_1)\| \cdots \|f'_{16}(x'_{16})$ is computed (where $x' = x'_1 \cdots x'_{16}$), using possibly different functions f'_i, and the bits of the result are permuted to give output x''. The $\{f_i\}$, $\{f'_i\}$, and the mixing permutation(s) could be random and dependent on the key, as we have described above. In practice, however, they are specially designed and fixed, and the key is incorporated in a different way, as we will describe below.

Substitution-permutation networks. A substitution-permutation network (SPN) can be viewed as a direct implementation of the confusion-diffusion paradigm. The difference is that now the round functions have a particular form rather than being chosen from the set of all possible permutations on some domain. Specifically, rather than having (a portion of) the key k specify an arbitrary permutation f, we instead fix a public "substitution function" (i.e., permutation) S called an *S-box*, and then let k define the function f given by $f(x) = S(k \oplus x)$.

To see how this works concretely, consider an SPN with a 64-bit block length based on a collection of 8-bit (1-byte) S-boxes $S_1, \ldots, S_8$. (See Figure 6.3.) Evaluating the cipher proceeds in a series of rounds, where in each round we apply the following sequence of operations to the 64-bit input x of that round (the input to the first round is just the input to the cipher):

1. *Key mixing:* Set $x := x \oplus k$, where k is the current-round sub-key;
2. *Substitution:* Set $x := S_1(x_1)\| \cdots \|S_8(x_8)$, where x_i is the ith byte of x;
3. *Permutation:* Permute the bits of x to obtain the output of the round.

The output of each round is fed as input to the next round. After the last round there is a *final key-mixing step*, and the result is the output of the cipher. (By Kerckhoffs' principle, we assume the S-boxes and the mixing permutation(s) are public and known to any attacker. This means that without a final key-mixing step, the last substitution and permutation steps would offer no additional security since they do not depend on the key.) Figure 6.4 shows

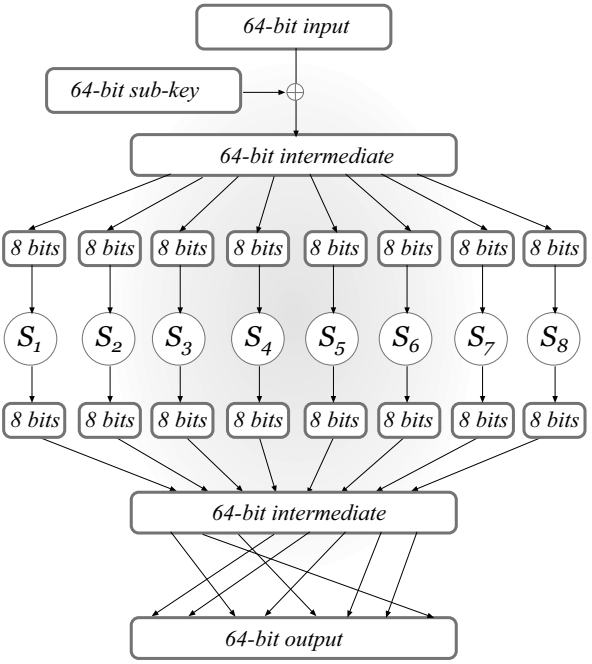


FIGURE 6.3: A single round of a substitution-permutation network.

the high-level structure of an SPN with a 16-bit block length and a different set of 4-bit S-boxes used in each round.

Different *sub-keys* (or *round keys*) are used in each round. The actual key of the block cipher is sometimes called the *master key*. The round sub-keys are derived from the master key according to a *key schedule*. The key schedule is often simple and may work by just taking different subsets of the bits of the master key, although more complex schedules can also be defined. An r-round SPN has r (full) rounds of key mixing, S-box substitution, and application of a mixing permutation, followed by a final key-mixing step. (This means that in an r-round SPN, $r + 1$ sub-keys are used.)

Any SPN is invertible (given the key). To see this, we show that given the output of the SPN and the key it is possible to recover the input. It suffices to show that a single round can be inverted; this implies the entire SPN can be inverted by working from the final round back to the beginning. But inverting a single round is easy: the mixing permutation can easily be inverted since it is just a re-ordering of bits. Since the S-boxes are permutations (i.e., one-to-one), these too can be inverted. The result can then be XORed with the appropriate sub-key to obtain the original input. Therefore:

PROPOSITION 6.3 *Let F be a keyed function defined by an SPN in which the S-boxes are all permutations. Then regardless of the key schedule and the number of rounds, F_k is a permutation for any k.*

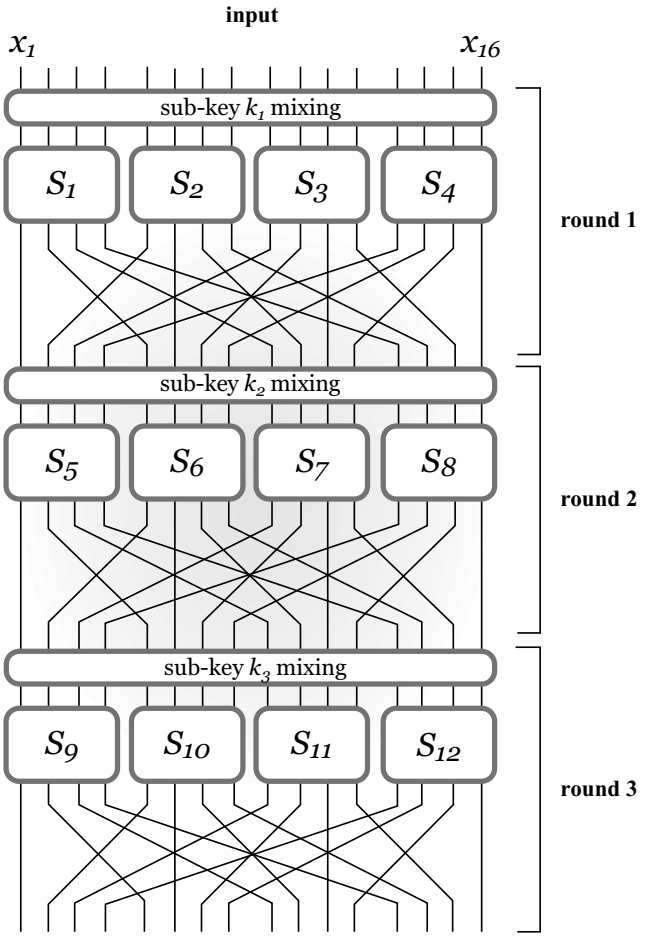


FIGURE 6.4: A substitution-permutation network.

The number of rounds, along with the exact choices of the S-boxes, mixing permutations, and key schedule, are what ultimately determine whether a given block cipher is trivially breakable or highly secure. We now discuss a basic principle behind the design of the S-boxes and mixing permutations.

The avalanche effect. As noted repeatedly, an important property in any block cipher is that a small change in the input must "affect" every bit of the output. We refer to this as the *avalanche effect*. One way to induce the avalanche effect in a substitution-permutation network is to ensure that the following two properties hold (and sufficiently many rounds are used):

1. The S-boxes are designed so that changing a single bit of the input to an S-box changes at least *two bits* in the output of the S-box.
2. The mixing permutations are designed so that the output bits of any given S-box are used as input to *multiple* S-boxes in the next round.

To see how this yields the avalanche effect, at least heuristically, assume that the S-boxes are all such that changing a single bit of the input of the S-box results in a change in exactly two bits of the output of the S-box, and that the mixing permutations are chosen as required above. For concreteness, assume the S-boxes have input/output size of 8 bits, and that the block length of the cipher is 128 bits. Consider now what happens when the block cipher is applied to two inputs that differ in a single bit:

1. After the first round, the intermediate values differ in exactly two bit-positions. This is because XORing the current sub-key maintains the 1-bit difference in the intermediate values, and so the inputs to all the S-boxes except one are identical. In the one S-box where the inputs differ, the output of the S-box causes a 2-bit difference. The mixing permutation applied to the results changes the positions of these differences, but maintains a 2-bit difference.

2. The mixing permutation applied at the end of the first round spreads the two bit-positions where the intermediate results differ into two *different* S-boxes in the second round. This remains true even after the appropriate sub-key is XORed with the result of the previous round. So, in the second round there are now *two* S-boxes that receive inputs differing by a single bit. Thus, at the end of the second round the intermediate values differ in 4 bits.

3. Continuing the same argument, we expect 8 bits of the intermediate value to be affected after the 3rd round, 16 bits to be affected after the 4th round, and all 128 bits of the output to be affected at the end of the 7th round.

The last point is not quite precise and it is certainly possible that there will be fewer differences than expected at the end of some round. (In fact, this must be the case because the outputs should not differ in all their bits, either.) For this reason, it is customary to use many more than 7 rounds. However, the above analysis gives a *lower bound* on the number of rounds: if fewer than 7 rounds are used then there must be some set of output bits that are not affected by a single-bit change in the input, implying that it will be possible to distinguish the cipher from a random permutation.

One might expect that the "best" way to design S-boxes would be to choose them at random (subject to the restriction that they are permutations). Interestingly, this turns out not to be the case, at least if we want to satisfy the design criterion mentioned earlier. Consider the case of an S-box operating on 4-bit inputs and let x and x' be two distinct values. Let $y = S(x)$, and now consider choosing uniform $y' \neq y$ as the value of $S(x')$. There are 4 strings that differ from y in only 1 bit, and so with probability $4/15$ we will choose y' that does *not* differ from y in two or more bits. The problem is compounded when we consider all pairs of inputs that differ in a single bit.

We conclude based on this example that, as a general rule, the S-boxes must be designed carefully rather than being chosen blindly at random. Random S-boxes are also not good for defending against attacks like the ones we will show in Section 6.2.6.

If a block cipher should also be *strongly* pseudorandom, then the avalanche effect must also apply to its *inverse*. That is, changing a single bit of the output should affect every bit of the input. For this it is useful if the S-boxes are designed so that changing a single bit of the output of an S-box changes at least two bits of the input to the S-box. Achieving the avalanche effect in both directions is another reason for further increasing the number of rounds.

Attacking Reduced-Round SPNs

Experience, along with many years of cryptanalytic effort, indicate that substitution-permutation networks are a good choice for constructing pseudorandom permutations as long as care is taken in the choice of the S-boxes, the mixing permutations, and the key schedule. The Advanced Encryption Standard, described in Section 6.2.5, is similar in structure to the substitution-permutation network described above, and is widely believed to be a strong pseudorandom permutation.

The strength of a cipher F constructed in this way depends heavily on the number of rounds. In order to obtain more of an insight into substitution-permutation networks, we will demonstrate attacks on SPNs having very few rounds. These attacks are straightforward, but are worth seeing as they demonstrate conclusively why a large number of rounds is needed.

A trivial case. We first consider a trivial case where F consists of one full round and no final key-mixing step. We show that an adversary given only a *single* input/output pair (x, y) can easily learn the secret key k for which $y = F_k(x)$. The adversary begins with the output value y and then inverts the mixing permutation and the S-boxes. It can do this, as noted before, because the full specification of the mixing permutation and the S-boxes is public. The intermediate value that the adversary computes is exactly $x \oplus k$ (assuming, without loss of generality, that the master key is used as the sub-key in the only round of the network). Since the adversary also knows the input x, it can immediately derive the secret key k. This is therefore a complete break.

Although this is a trivial attack, it demonstrates that in any substitution-permutation network there is no security gained by performing S-box substitution or applying a mixing permutation after the final sub-key mixing.

Attacking a one-round SPN. Now we have one full round followed by a key-mixing step. For concreteness, we assume a 64-bit block length and S-boxes with 8-bit (1-byte) input/output length. We assume independent 64-bit sub-keys k_1, k_2 are used for the two key-mixing steps, and so the master key $k_1 \| k_2$ of the SPN is 128 bits long.

A first observation is that we can extend the attack from the trivial case above to give a key-recovery attack here using much less than 2^{128} work. The idea is as follows: Given a single input/output pair (x, y) as before, the attacker enumerates over all possible values for the second-round sub-key k_2. For each such value, the attacker can invert the final key-mixing step to get a candidate intermediate value y'. We have seen above that given an input x and an output y' of a (full) SPN round, a unique possible sub-key k_1 can be easily identified. Thus, for each possible choice of k_2 the attacker derives a unique corresponding k_1 for which $k_1 \| k_2$ can be the master key. In this way, the attacker obtains (in 2^{64} time) a list of 2^{64} possibilities for the master key. These can be narrowed down using additional input/output pairs in roughly 2^{64} additional time; see also below.

A better attack is possible by noting that individual bits of the output depend on only part of the master key. Fix some given input/output pair (x, y) as before. Now, the adversary will enumerate over all possible values for the *first byte* of k_2. It can XOR each such value with the first byte of y to obtain a candidate value for the output of the first S-box. Inverting this S-box, the attacker learns a candidate value for the *input* to that S-box. Since the input to that S-box is the XOR of 8 bits of x and 8 bits of k_1 (where the positions of those bits depend on the first-round mixing permutation and are known to the attacker), this yields a candidate value for 8 bits of k_1.

To summarize: for each candidate value of the first byte of k_2, there is a *unique* possible corresponding value for some 8 bits of k_1. Put differently, this means that for some 16 bits of the master key, the attacker has reduced the number of possible values for those bits from 2^{16} to 2^8. The attacker can tabulate all those feasible values in 2^8 time. This can be repeated for each byte of k_2, giving 8 lists—each containing 2^8 values—that together characterize the possible values of the entire master key. The attacker has thus reduced the number of possible master keys to $(2^8)^8 = 2^{64}$, as in the earlier attack. The total time to do this, however, is now $8 \cdot 2^8 = 2^{11}$, a dramatic improvement.

The attacker can use additional input/output pairs to further reduce the space of possible keys. Consider the list of 2^8 feasible values for some set of 16 bits of the master key. The attacker knows that the correct value from that list must be consistent with any additional input/output pairs the attacker learns. Heuristically, any *incorrect* value from the list is consistent with some additional input/output pair (x', y') with probability no better than random guessing; since each 16-bit value from the table can be used to compute 1 byte of the output given the input x', we expect that an incorrect value will be consistent with the actual output y' with probability 2^{-8}. A small number of additional input/output pairs will thus suffice to narrow down *all* the tables to just a single value each, at which point the entire master key is known.

There is an important lesson to be learned here. The attack is possible since different parts of the key can be isolated from other parts. Thus, further *diffusion* is needed to make sure that all the bits of the key affect all of the bits of the output. Multiple rounds are needed for this to take place.

Attacking a two-round SPN. It is possible to extend the above ideas to give a better-than-brute-force attack on a two-round SPN using independent sub-keys in each round; we leave this as an exercise.

Instead, we simply note that a two-round SPN will not be a good pseudorandom permutation. Here we rely on the fact, mentioned earlier, that the avalanche effect does not occur after only two rounds (of course, this depends on the block length of the cipher and the input/output length of the S-boxes, but with reasonable parameters this will be the case). An attacker can distinguish a two-round SPN from a uniform permutation if it learns the result of evaluating the SPN on two inputs that differ in a single bit: in a two-round SPN many bits of the two outputs will be the same, something not expected to occur for a random permutation.)

6.2.2 Feistel Networks

Feistel networks offer another approach for constructing block ciphers. An advantage of Feistel networks over substitution-permutation networks is that the underlying functions used in a Feistel network—in contrast to the S-boxes used in SPNs—need not be invertible. *A Feistel network thus gives a way to construct an invertible function from non-invertible components.* This is important because a good block cipher should have "unstructured" behavior (so it looks random), yet requiring all the components of a construction to be invertible inherently introduces structure. Requiring invertibility also introduces an additional constraint on S-boxes, making them harder to design.

A Feistel network operates in a series of rounds. In each round, a keyed *round function* is applied in the manner described below. Round functions need not be invertible. They will typically be constructed from components like S-boxes and mixing permutations, but a Feistel network can deal with *any* round functions irrespective of their design.

In a balanced Feistel network (the only type we will consider), the ith round function $\hat{f}_i$ takes as input a sub-key k_i and an $\ell/2$-bit string and outputs an $\ell/2$-bit string. As in the case of SPNs, a master key k is used to derive sub-keys for each round. When some master key is chosen, thereby determining each sub-key k_i, we define $f_i : \{0,1\}^{\ell/2} \to \{0,1\}^{\ell/2}$ via $f_i(R) \stackrel{\text{def}}{=} \hat{f}_i(k_i, R)$. Note that the round functions $\hat{f}_i$ are fixed and publicly known, but the f_i depend on the master key and so are not known to the attacker.

The ith round of a Feistel network operates as follows. The input to the round is divided into two halves denoted L_{i-1} and R_{i-1} (the "left" and "right" halves, respectively). If the block length of the cipher is ℓ bits, then L_{i-1} and R_{i-1} each has length $\ell/2$. The output (L_i, R_i) of the round is

$$L_i := R_{i-1} \quad \text{and} \quad R_i := L_{i-1} \oplus f_i(R_{i-1}). \tag{6.3}$$

In an r-round Feistel network, the ℓ-bit input to the network is parsed as (L_0, R_0), and the output is the ℓ-bit value (L_r, R_r) obtained after applying all r rounds. A three-round Feistel network is shown in Figure 6.5.

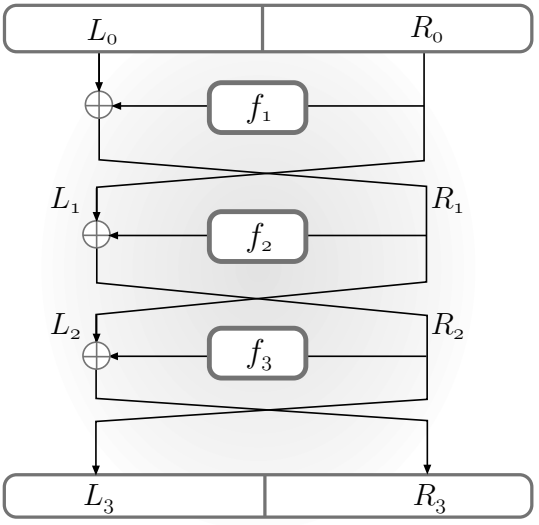


FIGURE 6.5: A three-round Feistel network.

Inverting a Feistel network. A Feistel network is invertible *regardless of the* $\{f_i\}$ (and thus regardless of the round functions $\{\hat{f}_i\}$). To show this we need only show that each round of the network can be inverted if the $\{f_i\}$ are known. Given the output (L_i, R_i) of the ith round, we can compute (L_{i-1}, R_{i-1}) as follows: first set $R_{i-1} := L_i$. Then compute

$$L_{i-1} := R_i \oplus f_i(R_{i-1}).$$

This gives the value (L_{i-1}, R_{i-1}) that was the input of this round (i.e., it computes the inverse of Equation (6.3)). Note that f_i is evaluated only in the forward direction, so it need not be invertible. We thus have:

PROPOSITION 6.4 *Let F be a keyed function defined by a Feistel network. Then regardless of the round functions $\{\hat{f}_i\}$ and the number of rounds, F_k is an efficiently invertible permutation for all k.*

As in the case of substitution-permutation networks, attacks on Feistel networks are possible when the number of rounds is too low. We will see such attacks when we discuss DES in the next section. Theoretical results concerning the security of Feistel networks are discussed in Section 7.6.

6.2.3 DES – The Data Encryption Standard

The Data Encryption Standard, or DES, was developed in the 1970s by IBM (with help from the National Security Agency) and adopted in 1977 as

a Federal Information Processing Standard for the US. In its basic form, DES is no longer considered secure due to its short key length of 56 bits, which makes it vulnerable to brute-force attacks. Nevertheless, it remains in wide use today in the strengthened form of triple-DES, described in Section 6.2.4.

DES is of great historical significance. It has undergone intensive scrutiny within the cryptographic community, arguably more than any other cryptographic algorithm in history. The common consensus is that, apart from its key length, DES is an extremely well-designed cipher. Indeed, even after many years, the best known attack on DES *in practice* is an exhaustive search over all 2^{56} possible keys. (As we will see, there are important theoretical attacks on DES that require less computation; however, these attacks assume certain conditions that seem difficult to realize in practice.)

In this section, we provide a high-level overview of the main components of DES. We stress that we will not provide a full specification that is correct in every detail, and some parts of the design will be omitted from our description. Our aim is to present the basic ideas underlying the construction of DES, and not all the low-level details; the reader interested in such details can consult the references at the end of this chapter.

The Design of DES

The DES block cipher is a 16-round Feistel network with a block length of 64 bits and a key length of 56 bits. The same round function $\hat{f}$ is used in each of the 16 rounds. The round function takes a 48-bit sub-key and, as expected for a (balanced) Feistel network, a 32-bit input (namely, half a block). The *key schedule* of DES is used to derive a sequence of 48-bit sub-keys $k_1, \ldots, k_{16}$ from the 56-bit master key. The key schedule of DES is relatively simple, with each sub-key k_i being a permuted subset of 48 bits of the master key. For our purposes, it suffices to note that the 56 bits of the master key are divided into two halves—a "left half" and a "right half"—each containing 28 bits. (This division occurs after an initial permutation is applied to the key, but we ignore this in our description.) In each round, the left-most 24 bits of the sub-key are taken as some subset of the 28 bits in the left half of the master key, and the right-most 24 bits of the round sub-key are taken as some subset of the 28 bits in the right half of the master key. We stress that the entire key schedule (including the manner in which bits are divided into the left and right halves, and which bits are used in forming each sub-key k_i) is fixed and public, and the only secret is the master key itself.

The DES round function. The DES round function $\hat{f}$—sometimes called the *DES mangler function*—is constructed using a paradigm we have previously analyzed: it is (essentially) just a substitution-permutation network! In more detail, computation of $\hat{f}(k_i, R)$ with $k_i \in \{0,1\}^{48}$ and $R \in \{0,1\}^{32}$ proceeds as follows: first, R is *expanded* to a 48-bit value R'. This is carried out by simply duplicating half the bits of R; we denote this by $R' := E(R)$ where

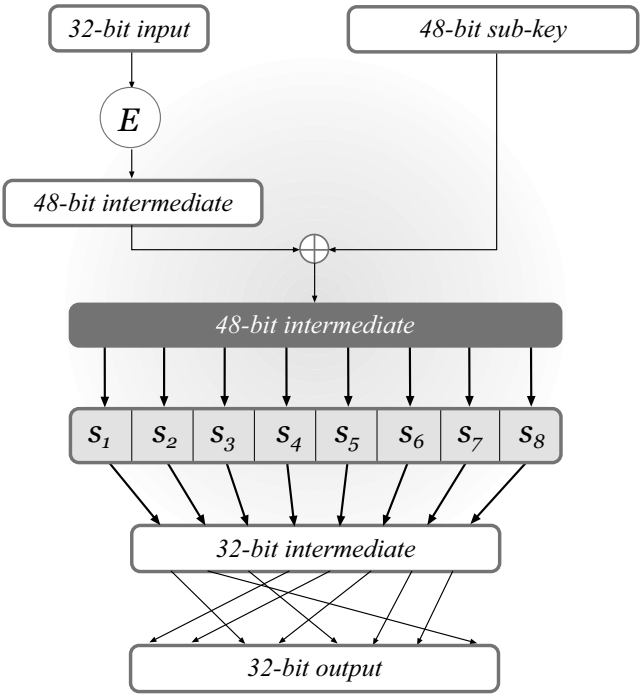


FIGURE 6.6: The DES mangler function.

E is called the *expansion function*. Following this, computation proceeds exactly as in our earlier discussion of SPNs: The expanded value R' is XORed with k_i, which is also 48 bits long, and the resulting value is divided into 8 blocks, each of which is 6 bits long. Each block is passed through a (different) S-box that takes a 6-bit input and yields a 4-bit output; concatenating the output from the 8 S-boxes gives a 32-bit result. A mixing permutation is then applied to the bits of this result to obtain the final output. See Figure 6.6.

One difference as compared to our original discussion of SPNs is that the S-boxes here are *not* invertible; indeed, they cannot be invertible since their inputs are longer than their outputs. Further discussion regarding the structural details of the S-boxes is given below.

We stress once again that everything in the above description (including the S-boxes themselves as well as the mixing permutation) is *publicly known*. The only secret is the master key which is used to derive all the sub-keys.

The S-boxes and the mixing permutation. The eight S-boxes that form the "core" of $\hat{f}$ are a crucial element of the DES construction and were very carefully designed. Studies of DES have shown that if the S-boxes were slightly modified, DES would have been much more vulnerable to attack.

This should serve as a warning to anyone who wishes to design a block cipher: seemingly arbitrary choices are not arbitrary at all, and if not made correctly may render the entire construction insecure.

Recall that each S-box maps a 6-bit input to a 4-bit output. Each S-box can be viewed as a table with 4 rows and 16 columns, where each cell of the table contains a 4-bit entry. A 6-bit input can be viewed as indexing one of the $2^6 = 64 = 4 \times 16$ cells of the table in the following way: The first and last input bits are used to choose the table row, and bits 2–5 are used to choose the table column. The 4-bit entry at some position of the table represents the output value for the input associated with that position.

The DES S-boxes have the following properties (among others):

1. Each S-box is a 4-to-1 function. (That is, exactly 4 inputs are mapped to each possible output.) This follows from the properties below.
2. Each row in the table contains each of the 16 possible 4-bit strings exactly once.
3. Changing *one bit* of any input to an S-box always changes at least *two bits* of the output.

The mixing permutation was also designed carefully. In particular it has the property that the four output bits from any S box will affect the input to six S-boxes in the next round. (This is possible because of the expansion function that is applied in the next round before the S-boxes are computed.)

The DES avalanche effect. The design of the mangler function ensures that DES exhibits a strong avalanche effect. In order to see this, we will trace the difference between the intermediate values in a DES computation of two inputs that differ by just a single bit. Let us denote the two inputs to the cipher by (L_0, R_0) and (L'_0, R'_0), where we assume that $R_0 = R'_0$ and so the single-bit difference occurs in the left half of the inputs (it may help to refer to Equation (6.3) and Figure 6.6 in what follows). After the first round the intermediate values (L_1, R_1) and (L'_1, R'_1) still differ by only a single bit, although now this difference is in the right half. In the second round of DES, the right half of each input is run through $\hat{f}$. Assuming that the bit where R_1 and R'_1 differ is not duplicated in the expansion step, the intermediate values before applying the S-boxes still differ by only a single bit. By property 3 of the S-boxes, the intermediate values *after* the S-box computation differ in at least *two* bits. The result is that the intermediate values (L_2, R_2) and (L'_2, R'_2) differ in *three* bits: there is a 1-bit difference between L_2 and L'_2 (carried over from the difference between R_1 and R'_1) and a 2-bit difference between R_2 and R'_2.

The mixing permutation spreads the two-bit difference between R_2 and R'_2 such that, in the following round, each of the two bits is used as input to a *different* S-box, resulting in a difference of at least 4 bits in the right halves of the intermediate values. (If either or both of the two bits in which R_2 and R'_2

differ are duplicated by E, the difference may be even greater.) There is also now a 2-bit difference in the left halves. As with a substitution-permutation network, we have an exponential effect and so after 7 rounds we expect all 32 bits in the right half to be affected (and after 8 rounds all 32 bits in the left half will be affected as well).

DES has 16 rounds, and so the avalanche effect occurs very early in the computation. This ensures that the computation of DES on similar inputs yields independent-looking outputs.

Attacks on Reduced-Round DES

A useful exercise for understanding more about the DES construction and its security is to look at the behavior of DES with only a few rounds. We will show attacks on one-, two-, and three-round variants of DES (recall that the real DES has 16 rounds). DES with three rounds or fewer cannot be a pseudorandom function because three rounds are not enough for the avalanche effect to occur. Thus, we will be interested in demonstrating more difficult (and more damaging) key-recovery attacks which compute the key k using only a relatively small number of input/output pairs computed using that key. Some of the attacks are similar to those we have seen in the context of substitution-permutation networks; here, however, we will see how they are applied to a concrete block cipher rather than to an abstract design.

The attacks below will be known-plaintext attacks in which the adversary knows some plaintext/ciphertext pairs $\{(x_i, y_i)\}$ with $y_i = DES_k(x_i)$ for some secret key k. When we describe the attacks, we will focus on a particular input/output pair (x, y) and will describe the information about the key that the adversary can derive from this pair. Continuing to use the notation developed earlier, we denote the left and right halves of the input x as L_0 and R_0, respectively, and let L_i, R_i denote the left and right halves after the ith round. Recall that E denotes the DES expansion function, k_i denotes the sub-key used in round i, and $f_i(R) = \hat{f}(k_i, R)$ denotes the actual function being applied in the Feistel network in the ith round.

One-round DES. Say we are given an input/output pair (x, y). In one-round DES, we have $y = (L_1, R_1)$, where $L_1 = R_0$ and $R_1 = L_0 \oplus f_1(R_0)$. We therefore know an input/output pair for f_1; specifically, we know that $f_1(R_0) = R_1 \oplus L_0$. By applying the inverse of the mixing permutation to the output $R_1 \oplus L_0$, we obtain the intermediate value consisting of the outputs from all the S-boxes, where the first 4 bits are the output from the first S-box, the next 4 bits are the output from the second S-box, and so on.

Consider the (known) 4-bit output of the first S-box. Since each S-box is a 4-to-1 function, this means there are exactly four possible inputs to this S-box that would result in the given output, and similarly for all the other S-boxes; each such input is 6 bits long. The input to the S-boxes is simply the XOR of $E(R_0)$ with the sub-key k_1. Since R_0, and hence $E(R_0)$, is known, we can

compute a set of four possible values for each 6-bit portion of k_1. This means we have reduced the number of possible keys k_1 from 2^{48} to $4^{48/6} = 4^8 = 2^{16}$ (since there are four possibilities for each of the eight 6-bit portions of k_1). This is already a small number and so we can just try all the possibilities on a different input/output pair (x', y') to find the right key. We thus obtain the key using only two known plaintexts in time roughly 2^{16}.

Two-round DES. In two-round DES, the output y is equal to (L_2, R_2) where

$$\begin{aligned} L_1 &= R_0 \\ R_1 &= L_0 \oplus f_1(R_0) \\ L_2 &= R_1 = L_0 \oplus f_1(R_0) \\ R_2 &= L_1 \oplus f_2(R_1). \end{aligned}$$

L_0, R_0, L_2, and R_2 are known from the given input/output pair (x, y), and thus we also know $L_1 = R_0$ and $R_1 = L_2$. This means that we know the input/output of both f_1 and f_2, and so the same method used in the attack on one-round DES can be used here to determine both k_1 and k_2 in time roughly $2 \cdot 2^{16}$. This attack works even if k_1 and k_2 are completely independent keys, although in fact the key schedule of DES ensures that many of the bits of k_1 and k_2 are equal (which can be used to further speed up the attack).

Three-round DES. Referring to Figure 6.5, the output value y is now equal to (L_3, R_3). Since $L_1 = R_0$ and $R_2 = L_3$, the only unknown values in the figure are R_1 and L_2 (which are equal).

Now we no longer have the input/output to any round function f_i. For example, the output value of f_2 is equal to $L_1 \oplus R_2$, where both of these values are known. However, we do *not* know the value R_1 that is input to f_2. Similarly, we can determine the inputs to f_1 and f_3 but not the outputs of those functions. Thus, the attack we used to break one-round and two-round DES will not work here.

Instead of relying on full knowledge of the input and output of one of the round functions, we will use knowledge of a certain relation between the inputs and outputs of f_1 and f_3. Observe that the output of f_1 is equal to $L_0 \oplus R_1 = L_0 \oplus L_2$, and the output of f_3 is equal to $L_2 \oplus R_3$. Therefore,

$$f_1(R_0) \oplus f_3(R_2) = (L_0 \oplus L_2) \oplus (L_2 \oplus R_3) = L_0 \oplus R_3,$$

where both L_0 and R_3 are known. That is, *the XOR of the outputs of f_1 and f_3* is known. Furthermore, the input to f_1 is R_0 and the input to f_3 is L_3, both of which are known. We conclude that we can determine the inputs to f_1 and f_3, and the XOR of their outputs. We now describe an attack that finds the secret key based on this information.

Recall that the key schedule of DES has the property that the master key is divided into a "left half," which we denote by k_L, and a "right half" k_R, each containing 28 bits. Furthermore, the 24 left-most bits of the sub-key used in

each round are taken only from k_L, and the 24 right-most bits of each sub-key are taken only from k_R. This means that k_L affects only the inputs to the first four S-boxes in any round, while k_R affects only the inputs to the last four S-boxes. Since the mixing permutation is known, we also know which bits of the output of each round function come out of each S-box.

The idea behind the attack is to separately traverse the key space for each half of the master key, giving an attack with complexity roughly $2 \cdot 2^{28}$ rather than complexity 2^{56}. Such an attack will be possible if we can verify a guess of half the master key, and we now show how this can be done. Say we guess some value for k_L, the left half of the master key. We know the input R_0 of f_1, and so using our guess of k_L we can compute the input to the first four S-boxes. This means that we can compute half the output bits of f_1 (the mixing permutation spreads out the bits we know, but since the mixing permutation is known we know exactly which bits these are). Likewise, we can compute the same locations in the output of f_3 by using the known input L_3 to f_3 and the same guess for k_L. Finally, we can compute the XOR of these output values and check whether they match the appropriate bits in the known value of the XOR of the outputs of f_1 and f_3. If they are not equal, then our guess for k_L is incorrect. A correct guess for k_L will always pass this test, and so will not be eliminated, but an incorrect guess is expected to pass this test only with probability roughly 2^{-16} (since we check equality of 16 bits in two computed values). There are 2^{28} possible values for k_L, so if each incorrect value remains a viable candidate with probability 2^{-16} then we expect to be left with only $2^{28} \cdot 2^{-16} = 2^{12}$ possibilities for k_L after the above.

By performing the above for each half of the master key, we obtain in time $2 \cdot 2^{28}$ approximately 2^{12} candidates for the left half and 2^{12} candidates for the right half. Since each combination of the left half and right half is possible, we have 2^{24} candidate keys overall and can run a brute-force search over this set using an additional input/output pair (x', y'). (An alternative approach which is more efficient is to simply repeat the previous attack using the 2^{12} remaining candidates for each half-key.) The time complexity of the attack is roughly $2 \cdot 2^{28} + 2^{24} < 2^{30}$, which is much less than 2^{56}.

Security of DES

After almost 30 years of intensive study, the best known practical attack on DES is still an exhaustive search through its key space. (We discuss some important theoretical attacks in Section 6.2.6. These attacks require a large number of input/output pairs, which would be difficult to obtain in an attack on any real-world system using DES.) Unfortunately, the 56-bit key length of DES is short enough that an exhaustive search through all 2^{56} possible keys is now feasible. Already in the late 1970s there were strong objections to the choice of such a short key for DES. Back then, the objection was theoretical, as the computational power needed to search through that many

keys was generally unavailable.[4] The practicality of a brute-force attack on DES, however, was demonstrated in 1997 when the first of a set of DES challenges set up by RSA Security was solved by the DESCHALL project using thousands of computers coordinated across the Internet; the computation took 96 days. A second challenge was broken the following year in just 41 days by the `distributed.net` project. A significant breakthrough came in 1998 when the third challenge was solved in just *56 hours*. This impressive feat was achieved via a special-purpose DES-breaking machine called *Deep Crack* that was built by the Electronic Frontier Foundation at a cost of $250,000. In 1999, a DES challenge was solved in just over 22 hours as a combined effort of Deep Crack and `distributed.net`. The current state-of-the-art is the DES cracking box by PICO Computing, which uses 48 FPGAs and can find a DES key in approximately 23 hours.

The time/space tradeoffs discussed in Section 5.4.3 show that exhaustive key-search attacks can be accelerated using pre-computation and additional memory. Due to the short key length of DES, time/space tradeoffs can be especially effective. Specifically, using pre-processing it is possible to generate a table a few terabytes large that then enables recovery of a DES key with high probability from a single input/output pair using approximately 2^{38} DES evaluations (which can be computed in mere minutes). The bottom line is that DES has a key that is far too short, and cannot be considered secure for any serious application today.

A secondary cause for concern is the relatively short block length of DES. A short block length is problematic because the concrete security of many constructions based on block ciphers depends on the block length—*even if the cipher used is "perfect."* For example, the proof of security for CTR mode (cf. Theorem 3.32) shows that even when a completely random function is used an attacker can break the security of this encryption scheme with probability $2q^2/2^\ell$ if it obtains q plaintext/ciphertext pairs. In the case of DES where $\ell = 64$, this means that if an attacker obtains only $q = 2^{30}$ plaintext/ciphertext pairs, security is compromised with high probability. Obtaining plaintext/ciphertext pairs is relatively easy if an adversary eavesdrops on the encryption of messages containing known headers, redundancies, etc.

The insecurity of DES has nothing to do with its design *per se*, but rather is due to its short key length (and, to a lesser extent, its short block length). This is a great tribute to the designers of DES, who seem to have succeeded in constructing an almost "perfect" block cipher (besides its too-short key). Since DES itself seems not to have significant structural weaknesses, it makes sense to use DES as a building block for constructing block ciphers with longer keys. We discuss this further in Section 6.2.4.

[4] In 1977, it was estimated that a computer that could crack DES in one day would cost $20 million to build.

The replacement for DES—the *Advanced Encryption Standard* (AES), covered later in this chapter—was explicitly designed to address concerns regarding the short key length and block length of DES. AES supports 128-, 192-, or 256-bit keys, and a block length of 128 bits.

Better-than-brute-force attacks on DES were first shown in the early 1990s by Biham and Shamir, who developed a technique called *differential cryptanalysis*. Their attack takes time 2^{37} and requires 2^{47} chosen plaintexts. While the attack was a breakthrough from a theoretical standpoint, it does not appear to be of much practical concern since it is hard to imagine a realistic scenario where an adversary can obtain this many encryptions of chosen plaintexts.

Interestingly, the work of Biham and Shamir indicated that the DES S-boxes had been specifically designed to be resistant to differential cryptanalysis, suggesting that the technique of differential cryptanalysis was known (but not publicly revealed) by the designers of DES. After Biham and Shamir announced their result, this suspicion was confirmed.

Linear cryptanalysis was developed by Matsui in the mid-1990s and was also applied successfully to DES. The advantage of this attack is that it uses *known* plaintexts rather than *chosen* plaintexts. Nevertheless, the number of plaintext/ciphertext pairs required—about 2^{43}—is still huge.

We briefly describe differential and linear cryptanalysis in Section 6.2.6.

6.2.4 3DES: Increasing the Key Length of a Block Cipher

The main weakness of DES is its short key. It thus makes sense to try to design a block cipher with a larger key length using DES as a building block. Some approaches to doing so are discussed in this section. Although we refer to DES frequently throughout the discussion, and DES is the most prominent block cipher to which these techniques have been applied, everything we say here applies generically to *any* block cipher.

Internal modifications vs. "black-box" constructions. There are two general approaches one could take to constructing another cipher based on DES. The first approach would be to somehow modify the *internal structure* of DES, while increasing the key length. For example, one could leave the round function untouched and simply use a 128-bit master key with a different key schedule (still choosing a 48-bit sub-key in each round). Or, one could change the S-boxes themselves and use a larger sub-key in each round. The disadvantage of such approaches is that by modifying DES—in even the smallest way—we lose the confidence we have gained in DES by virtue of the fact that it has remained resistant to attack for so many years. Cryptographic constructions are very sensitive, and even mild, seemingly insignificant changes can render a construction completely insecure.[5] Changing the internals of a block cipher is therefore not recommended.

[5] In fact, various results to this effect have been shown for DES; e.g., changing the S-boxes or the mixing permutation can make DES much more vulnerable to attack.

An alternative approach that does not suffer from the above problem is to use DES as a "black box" and not touch its internal structure at all. In this approach we treat DES as a "perfect" block cipher with a 56-bit key, and construct a new block cipher that only invokes the original, unmodified DES. Since DES itself is not tampered with, this is a much more prudent approach, and is the one we will pursue here.

Double Encryption

Let F be a block cipher with an n-bit key length and ℓ-bit block length. Then a new block cipher F' with a key of length $2n$ can be defined by

$$F'_{k_1,k_2}(x) \stackrel{\text{def}}{=} F_{k_2}(F_{k_1}(x)),$$

where k_1 and k_2 are independent keys. For the case where F is DES, we obtain a cipher F' called 2DES that takes a 112-bit key; if exhaustive key search were the best available attack, a key length of 112 bits would be sufficient since an attack requiring time 2^{112} is completely out of reach. Unfortunately, we now show an attack on F' that runs in time roughly 2^n, significantly less than the 2^{2n} time one would hope would be necessary to carry out an exhaustive search for a $2n$-bit key. This means that the new block cipher is essentially no better than the old one, even though it has a key that is twice as long.[6]

The attack is called a "meet-in-the-middle attack," for reasons that will soon become clear. Say the adversary is given a single input/output pair (x, y), where $y = F'_{k_1^*,k_2^*}(x) = F_{k_2^*}(F_{k_1^*}(x))$ for unknown k_1^*, k_2^*. The adversary can narrow down the set of possible keys in the following way:

1. For each $k_1 \in \{0,1\}^n$, compute $z := F_{k_1}(x)$ and store (z, k_1) in a list L.
2. For each $k_2 \in \{0,1\}^n$, compute $z := F^{-1}_{k_2}(y)$ and store (z, k_2) in a list L'.
3. Entries $(z_1, k_1) \in L$ and $(z_2, k_2) \in L'$ are a *match* if $z_1 = z_2$. For each such match, add (k_1, k_2) to a set S. (Matches can be found easily after sorting L and L' by their first components.)

See Figure 6.7 for a graphical depiction of the attack.

The attack takes time $\mathcal{O}(n \cdot 2^n)$, and requires space $\mathcal{O}((n+\ell) \cdot 2^n)$. The set S output by this algorithm contains exactly those values (k_1, k_2) for which

$$F_{k_1}(x) = F^{-1}_{k_2}(y) \tag{6.4}$$

or, equivalently, for which $y = F'_{k_1,k_2}(x)$. In particular, $(k_1^*, k_2^*) \in S$. On the other hand, a pair $(k_1, k_2) \neq (k_1^*, k_2^*)$ is (heuristically) expected to satisfy Equation (6.4) with probability $2^{-\ell}$ if we treat $F_{k_1}(x)$ and $F^{-1}_{k_2}(y)$ as uniform

[6] This is not quite true since a brute-force attack on F can be carried out in time 2^n and constant memory, whereas the attack we show on F' requires 2^n time *and* 2^n *memory*. Nevertheless, the attack illustrates that F' does not achieve the desired level of security.

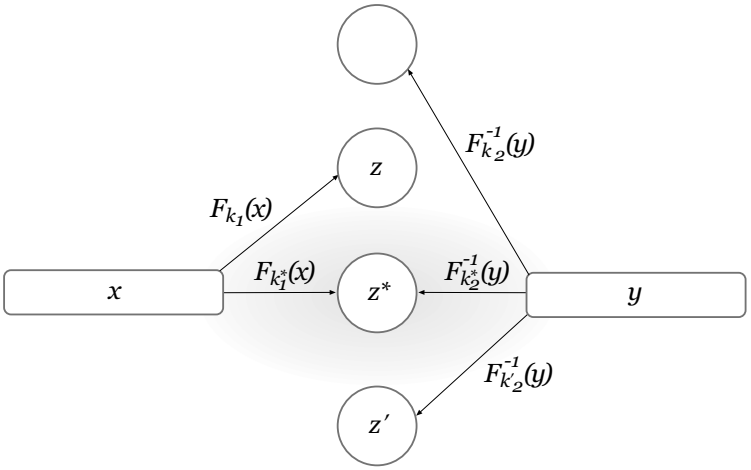


FIGURE 6.7: A meet-in-the-middle attack.

ℓ-bit strings, and so the expected size of S is $2^{2n} \cdot 2^{-\ell} = 2^{2n-\ell}$. Using another few input/output pairs, and taking the intersection of the sets that are obtained, the correct (k_1^*, k_2^*) can be identified with very high probability.

Triple Encryption

The obvious generalization of the preceding approach is to apply the block cipher *three* times in succession. Two variants of this approach are common:

Variant 1: three keys. Choose three independent keys k_1, k_2, k_3 and define $F''_{k_1,k_2,k_3}(x) \stackrel{\text{def}}{=} F_{k_3}(F_{k_2}^{-1}(F_{k_1}(x)))$.

Variant 2: two keys. Choose two independent keys k_1, k_2 and then define $F''_{k_1,k_2}(x) \stackrel{\text{def}}{=} F_{k_1}(F_{k_2}^{-1}(F_{k_1}(x)))$.

Before comparing the security of the two alternatives we note that the middle invocation of F is reversed. If F is a sufficiently good cipher this makes no difference as far as security is concerned, since if F is a strong pseudorandom permutation then F^{-1} must be too. The reason for reversing the second application of F is to obtain backward compatibility: if one sets $k_1 = k_2 = k_3$, the resulting function is equivalent to a single invocation of F using key k_1.

Security of the first variant. The key length of the first variant is $3n$, and so we might hope that the best attack on this cipher would require time 2^{3n}. However, the cipher is susceptible to a meet-in-the-middle attack just as in the case of double encryption, though the attack now takes time 2^{2n}. This is the best known attack. Thus, although this variant is not as secure as one might have hoped, it obtains sufficient security for all practical purposes even for $n = 56$ (assuming, of course, the original cipher F has no weaknesses).

Security of the second variant. The key length of this variant is $2n$ and so the best we can hope for is security against attacks running in time 2^{2n}. There is no known attack with better time complexity when the adversary is given only a small number of input/output pairs. (See Exercise 6.13 for an attack using 2^n chosen plaintexts.) Thus, two-key triple encryption is a reasonable choice in practice.

Triple-DES (3DES). Triple-DES (or 3DES) is based on a triple invocation of DES using two or three keys, as described above. 3DES was standardized in 1999, and is widely used today. Its main drawbacks are its relatively small block length and the fact that it is relatively slow since it requires 3 full block-cipher operations. Since the minimum recommended key length nowadays is 128 bits, 2-key 3DES is no longer recommended (due to its key length of only 112 bits). These drawbacks have led to the replacement of DES/triple-DES by the Advanced Encryption Standard, presented in the next section.

6.2.5 AES – The Advanced Encryption Standard

In January 1997, the United States National Institute of Standards and Technology (NIST) announced that it would hold a competition to select a new block cipher—to be called the *Advanced Encryption Standard*, or AES—to replace DES. The competition began with an open call for teams to submit candidate block ciphers for evaluation. A total of 15 different algorithms were submitted from all over the world, including contributions from many of the best cryptographers and cryptanalysts. Each team's candidate cipher was intensively analyzed by members of NIST, the public, and (especially) the other teams. Two workshops were held, one in 1998 and one in 1999, to discuss and analyze the various submissions. Following the second workshop, NIST narrowed the field down to 5 "finalists" and the second round of the competition began. A third AES workshop was held in April 2000, inviting additional scrutiny on the five finalists. In October 2000, NIST announced that the winning algorithm was Rijndael (a block cipher designed by the Belgian cryptographers Vincent Rijmen and Joan Daemen), although NIST conceded that any of the 5 finalists would have made an excellent choice. In particular, no serious security vulnerabilities were found in any of the 5 finalists, and the selection of a "winner" was based in part on properties such as efficiency, performance in hardware, flexibility, etc.

The process of selecting AES was ingenious because any group that submitted an algorithm, and was therefore interested in having its algorithm adopted, had strong motivation to find attacks on the other submissions. In this way, the world's best cryptanalysts focused their attention on finding even the slightest weaknesses in the candidate ciphers submitted to the competition. After only a few years each candidate algorithm was already subjected to intensive study, thus increasing our confidence in the security of the winning algorithm. Of course, the longer the algorithm is used and studied without

being broken, the more our confidence will continue to grow. Today, AES is widely used and no significant security weaknesses have been discovered.

The AES construction. In this section, we present the high-level structure of Rijndael/AES. (Technically speaking, Rijndael and AES are not the same thing but the differences are unimportant for our discussion here.) As with DES, we will not present a full specification and our description should not be used as a basis for implementation. Our aim is only to provide a general idea of how the algorithm works.

The AES block cipher has a 128-bit block length and can use 128-, 192-, or 256-bit keys. The length of the key affects the key schedule (i.e., the sub-key that is used in each round) as well as the number of rounds, but does not affect the high-level structure of each round.

In contrast to DES, which uses a Feistel structure, AES is essentially a substitution-permutation network. During computation of the AES algorithm, a 4-by-4 array of bytes called the *state* is modified in a series of rounds. The state is initially set equal to the input to the cipher (note that the input is 128 bits, which is exactly 16 bytes). The following operations are then applied to the state in a series of four stages during each round:

Stage 1 – AddRoundKey**:** In every round of AES, a 128-bit sub-key is derived from the master key, and is interpreted as a 4-by-4 array of bytes. The state array is updated by XORing it with this sub-key.

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 $\{0,1\}^8$.

Stage 3 – ShiftRows**:** In this step, the bytes in each row of the state array are shifted to the left as follows: the first row of the array is untouched, the second row is shifted one place to the left, the third row is shifted two places to the left, and the fourth row is shifted three places to the left. All shifts are cyclic so that, e.g., in the second row the first byte becomes the fourth byte.

Stage 4 – MixColumns**:** In this step, an invertible transformation is applied to the four bytes in each column. (Technically speaking, this is a linear transformation—i.e., matrix multiplication—over an appropriate field.) This transformation has the property that if two inputs differ in $b > 0$ bytes, then applying the transformation yields two outputs differing in at least $5 - b$ bytes.

In the final round, MixColumns is replaced with AddRoundKey. This prevents an adversary from simply inverting the last three stages, which do not depend on the key.

By viewing stages 3 and 4 together as a "mixing" step, we see that each round of AES has the structure of a substitution-permutation network: the

round sub-key is first XORed with the input to the current round; next, a small, invertible function is applied to "chunks" of the resulting value; finally, the bits of the result are mixed in order to obtain diffusion. The only difference is that, unlike our previous description of substitution-permutation networks, here the mixing step does not consist of a simple shuffling of the bits but is instead carried out using a shuffling plus an invertible linear transformation. (Simplifying things a bit and looking at a trivial 3-bit example, shuffling the bits of $x = x_1\|x_2\|x_3$ might, e.g., map x to $x' = x_2\|x_1\|x_3$. An invertible linear transformation might map x to $x_1 \oplus x_2\|x_2 \oplus x_3\|x_1 \oplus x_2 \oplus x_3$.)

The number of rounds depends on the key length. Ten rounds are used for a 128-bit key, 12 rounds for a 192-bit key, and 14 rounds for a 256-bit key.

Security of AES. As we have mentioned, the AES cipher was subject to intense scrutiny during the selection process and has continued to be studied ever since. To date, there are no practical cryptanalytic attacks that are significantly better than an exhaustive search for the key.

We conclude that, as of today, AES constitutes an excellent choice for any cryptographic scheme that requires a (strong) pseudorandom permutation. It is free, standardized, efficient, and highly secure.

6.2.6 *Differential and Linear Cryptanalysis

Block ciphers are relatively complicated, and as such are difficult to analyze. Nevertheless, one should not be fooled into thinking that a complicated cipher is difficult to break. On the contrary, it is very hard to construct a secure block cipher and surprisingly easy to find attacks on most constructions (no matter how complicated they appear). This should serve as a warning that non-experts should not try to construct new ciphers. Given the availability of triple-DES and AES, it is hard to justify using anything else.

In this section we describe two tools that are now a standard part of the cryptanalyst's toolbox. Our goal here is give a taste of some advanced cryptanalysis, as well as to reinforce the idea that designing a secure block cipher involves careful choice of its components.

Differential cryptanalysis. This technique, which can lead to a chosen-plaintext attack on a block cipher, was first presented in the late 1980s by Biham and Shamir, who used it to attack DES in 1993. The basic idea behind the attack is to tabulate *specific differences in the input* that lead to *specific differences in the output* with probability greater than would be expected for a random permutation. Specifically, say *the differential* (Δ_x, Δ_y) *occurs in some keyed permutation* F' *with probability* p if for uniform inputs x_1 and x_2 satisfying $x_1 \oplus x_2 = \Delta_x$, and uniform choice of key k, the probability that $F'_k(x_1) \oplus F'_k(x_2) = \Delta_y$ is p. For any fixed (Δ_x, Δ_y) and x_1, x_2 satisfying $x_1 \oplus x_2 = \Delta_x$, if we choose a uniform function $f : \{0,1\}^\ell \to \{0,1\}^\ell$, we have $\Pr[f(x_1) \oplus f(x_2) = \Delta_y] = 2^{-\ell}$. In a weak block cipher, however, there may

be differentials that occur with significantly higher probability. This can be leveraged to give a full key-recovery attack, as we now show for SPNs.

We describe the basic idea, and then work through a concrete example. Let F be a block cipher with ℓ-bit block length which is an r-round SPN, and let $F'_k(x)$ denote the intermediate result in a computation of $F_k(x)$ after applying the key-mixing step of round r. (That is, F' excludes the S-box substitution and mixing permutation of the last round, as well as the final key-mixing step.) Say there is a differential (Δ_x, Δ_y) in F' that occurs with probability $p \gg 2^{-\ell}$. It is possible to exploit this high-probability differential to learn bits of the final mixing sub-key k_{r+1}. The high-level idea is as follows: let $\{(x_1^i, x_2^i)\}_{i=1}^L$ be a collection of L pairs of random inputs with differential Δ_x, i.e., with $x_1^i \oplus x_2^i = \Delta_x$ for all i. Using a chosen-plaintext attack, obtain the values $y_1^i = F_k(x_1^i)$ and $y_2^i = F_k(x_2^i)$ for all i. Now, for all possible bitstrings $k^* \in \{0,1\}^\ell$, compute $\tilde{y}_1^i = F'_k(x_1^i)$ and $\tilde{y}_2^i = F'_k(x_2^i)$, assuming the value of the final sub-key k_{r+1} is k^*. This is done by inverting the final key-mixing step using k^*, and then inverting the mixing permutation and S-boxes of round r, which do not depend on the master key. When $k^* = k_{r+1}$, we expect that a p-fraction of the pairs will satisfy $\tilde{y}_1^i \oplus \tilde{y}_2^i = \Delta_y$. On the other hand, when $k^* \neq k_{r+1}$ we may heuristically expect only a $2^{-\ell}$-fraction of the pairs to yield this differential. By setting L large enough, the correct value of the final sub-key k_{r+1} can be determined.

This works, but is not very efficient since in each step we enumerate over 2^ℓ possible values. We can do better by guessing portions of k_{r+1} at a time. More concretely, assume the S-boxes in F have 1-byte input/output length, and focus on the first byte of Δ_y, which we assume is nonzero. It is possible to verify if the differential holds in that byte by guessing only 8 bits of k_{r+1}, namely, the 8 bits that correspond (after the round-r mixing permutation) to the output of the first S-box. Thus, proceeding as above, we can learn these 8 bits by enumerating over all possible values for those bits, and seeing which value yields the desired differential in the first byte with the highest probability. Incorrect guesses for those 8 bits yield the expected differential in that byte with (heuristic) probability 2^{-8}, but the correct guess will give the expected differential with probability roughly $p + 2^{-8}$; this is because with probability p the differential holds on the entire block (so in particular for the first byte), and when this is not the case then we can treat the differential in the first byte as random. Note that different differentials may be needed to learn different portions of k_{r+1}.

In practice, various optimizations are performed to improve the effectiveness of the above test or, more specifically, to increase the gap between the probability that an incorrect guess yields the differential vs. the probability that a correct guess does. One optimization is to use a *low-weight* differential in which Δ_y has many zero bytes in the positions that enter the S-boxes in round r. Any pairs $\tilde{y}_1, \tilde{y}_2$ satisfying such a differential have equal values entering many of the S-boxes in round r, and so will result in output values y_1, y_2 that are equal in the corresponding bit-positions (depending on the final

mixing permutation). This means that when performing the test described earlier, one can simply discard any pairs (y_1^i, y_2^i) that do not agree in those bit-positions (since the corresponding intermediate values $(\tilde{y}_1, \tilde{y}_2)$ cannot possibly satisfy the differential, for any choice of the final sub-key). This significantly improves the effectiveness of the attack.

Once k_{r+1} is known, the attacker can "peel off" the final key-mixing step, as well as the mixing permutation and S-box substitution steps of round r (since these do not depend on the master key), and then apply the same attack—using a different differential—to find the rth-round sub-key k_r, and so on, until it learns all sub-keys (or, equivalently, the entire master key).

A worked example. We work through a "toy" example, illustrating also how a good differential can be found. We use a four-round SPN with a block length of 16 bits, based on a single S-box with 4-bit input/output length. The S-box is defined as follows (the table shows how each 4-bit input is mapped to a 4-bit output):

Input:	0000	0001	0010	0011	0100	0101	0110	0111
Output:	0000	1011	0101	0001	0110	1000	1101	0100
Input:	1000	1001	1010	1011	1100	1101	1110	1111
Output:	1111	0111	0010	1100	1001	0011	1110	1010

The mixing permutation, showing where each bit is moved for each of the 16 bits in a block, is as follows:

In:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Out:	7	2	3	8	12	5	11	9	10	1	14	13	4	6	16	15

x	S(x)	x ⊕ 1111	S(x ⊕ 1111)	S(x) ⊕ S(x ⊕ 1111)
0000	0000	1111	1010	**1010**
0001	1011	1110	1110	0101
0010	0101	1101	0011	0110
0011	0001	1100	1001	1000
0100	0110	1011	1100	**1010**
0101	1000	1010	0010	**1010**
0110	1101	1001	0111	**1010**
0111	0100	1000	1111	1011
1000	1111	0111	0100	1011
1001	0111	0110	1101	**1010**
1010	0010	0101	1000	**1010**
1011	1100	0100	0110	**1010**
1100	1001	0011	0001	1000
1101	0011	0010	0101	0110
1110	1110	0001	1011	0101
1111	1010	0000	0000	**1010**

FIGURE 6.8: The effect of the input difference $\Delta_x = 1111$ in our S-box.

We first find a differential in the S-box. Let $S(x)$ denote the output of the S-box on input x. Consider the differential $\Delta_x = 1111$. Then, for example, we have $S(0000) \oplus S(1111) = 0000 \oplus 1010 = 1010$ and so in this case a difference of 1111 in the inputs leads to a difference of 1010 in the outputs. Let us see if this relation holds frequently. We have $S(0001) = 1011$ and $S(0001 \oplus 1111) = S(1110) = 1110$, and so here a difference of 1111 in the inputs does *not* lead to a difference of 1010 in the outputs. However, $S(0100) = 0110$ and $S(0100 \oplus 1111) = S(1011) = 1100$ and so in this case, a difference of 1111 in the inputs yields a difference of 1010 in the outputs. In Figure 6.8 we have tabulated results for all possible inputs. We see that *half* the time a difference of 1111 in the inputs yields a difference of 1010 in the outputs. Thus, $(1111, 1010)$ is a differential in S that occurs with probability $1/2$.

Output Difference Δ_y

Input Difference Δ_x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	4	0	0	0	2	2	2	2	0	0	4	0
2	0	0	0	0	0	2	0	2	0	2	2	4	2	2	0	0
3	0	2	2	4	0	4	0	0	0	0	0	0	0	2	2	0
4	0	0	0	2	2	2	6	0	2	0	0	0	2	0	0	0
5	0	2	2	0	0	0	0	0	4	0	0	0	4	2	2	0
6	0	2	0	2	0	0	0	0	0	2	0	2	0	4	0	4
7	0	2	0	0	2	4	2	2	0	2	0	0	0	2	0	0
8	0	0	0	2	0	0	0	2	0	0	0	2	2	2	2	4
9	0	2	0	2	2	2	0	4	0	2	2	0	0	0	0	0
A	0	0	4	0	2	0	2	4	2	0	2	0	0	0	0	0
B	0	0	2	0	0	0	2	0	0	2	0	0	4	2	4	0
C	0	0	0	0	0	0	0	0	4	4	0	4	0	0	0	4
D	0	4	2	2	0	0	2	2	0	0	0	0	0	0	0	4
E	0	2	4	2	4	0	0	0	0	0	0	0	2	0	2	0
F	0	0	0	0	0	2	2	0	2	0	**8**	2	0	0	0	0

FIGURE 6.9: Differentials in our S-box.

This same process can be carried out for all 2^4 input differences Δ_x to calculate the probability of every differential. Namely, for each pair (Δ_x, Δ_y) we tabulate the number of 4-bit inputs x for which $S(x) \oplus S(x \oplus \Delta_x) = \Delta_y$. We have done this for our example S-box in Figure 6.9. (For conciseness we represent (Δ_x, Δ_y) using hexadecimal notation.) The table should be read as follows: entry (i, j) counts how many inputs with difference i map to outputs with difference j. Observe, for example, that there are 8 inputs with difference `0xF` $= 1111$ that map to output `0xA` $= 1010$, as we have shown above. This is the highest-probability differential (apart from the trivial differential $(0, 0)$). But there are other differentials of interest: an input difference of `0x4` $= 0100$ maps to an output difference of `0x6` $= 0110$ with probability $6/16 = 3/8$, and there are several differentials with probability $4/16 = 1/4$.

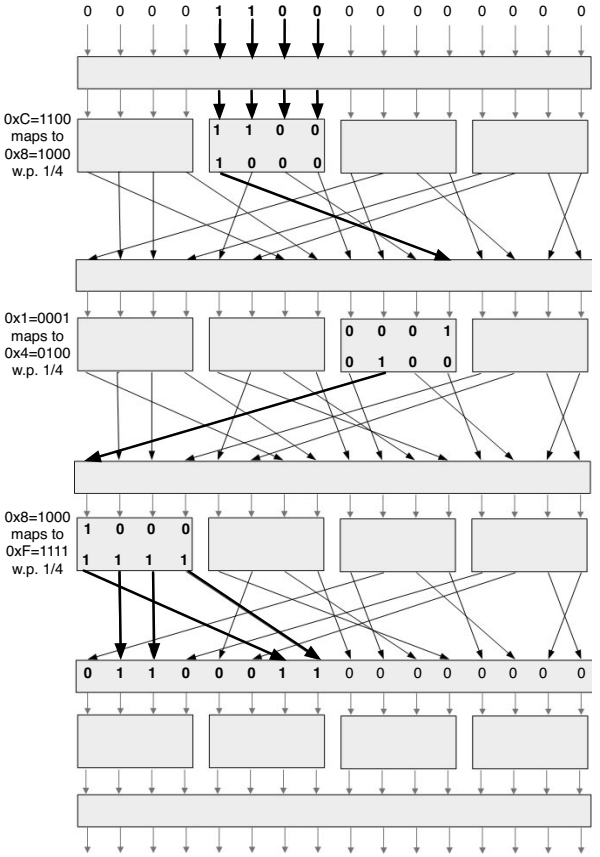


FIGURE 6.10: Tracing differentials through a four-round SPN that uses the S-box and mixing permutation given in the text.

We now extend this to find a good differential for the first three rounds of the SPN. Consider evaluating the SPN on two inputs that have a differential of 0000 1100 0000 0000, and tracing the differential between the intermediate values at each step of this evaluation. (Refer to Figure 6.10, which shows the four full rounds of the SPN plus the final key-mixing step. For clarity, the figure omits the mixing permutation in the 4th round; that mixing permutation just has the effect of shuffling the bits of the differential, and so can easily be taken into account in the attack.) The key-mixing step in the first round does not affect the differential, and so the inputs to the second S-box in the first round have differential 1100. We see from Figure 6.9 that a difference of `0xC` = 1100 in the inputs to the S-box yields a difference of `0x8` = 1000 in the outputs of the S-box with probability 1/4. So with probability 1/4 the differential in the output of the 2nd S-box after round 1 is a single bit which is

moved by the mixing permutation from the 5th position to the 12th position. (The inputs to the other S-boxes are equal, so their outputs are equal and the differential of the outputs is 0000.) Assuming this to be the case, the input difference to the third S-box in the second round is `0x1` = 0001 (once again, the key-mixing step in the second round does not affect the differential); using Figure 6.9 we have that with probability 1/4 the output difference from that S-box is `0x4` = 0100. Thus, once again there is just a single output bit that is different, and it is moved from the 10th position to the first position by the mixing permutation. Finally, consulting Figure 6.9 yet again, we see that an input difference of `0x8` = 1000 to the S-box results in an output difference of `0xF` = 1111 with probability 1/4. The bits in positions 1, 2, 3, and 4 are then moved by the mixing permutation to positions 7, 2, 3, and 8.

Overall, then, we see that an input difference of $\Delta_x = 0000\ 1100\ 0000\ 0000$ yields the output difference $\Delta_y = 0110\ 0011\ 0000\ 0000$ after three rounds with probability at least $\frac{1}{4} \cdot \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{64}$.[7] (We multiply the probabilities since we heuristically assume independence of each round.) For a random function, the probability that any given differential occurs is just $2^{-16} = 1/65536$. Thus, the differential we have found occurs with probability significantly higher than what would be expected for a random function. Observe also that we have found a low-weight differential.

We can use this differential to find the first 8 bits of the final sub-key k_5. As discussed earlier, we begin by letting $\{(x_1^i, x_2^i)\}_{i=1}^L$ be a set of L pairs of random inputs with differential Δ_x. Using a chosen-plaintext attack, we then obtain the values $y_1^i = F_k(x_1^i)$ and $y_2^i = F_k(x_2^i)$ for all i. Now, for all possible values for the initial 8 bits of k_5, we compute the initial 8 bits of $\tilde{y}_1^i, \tilde{y}_2^i$, the intermediate values after the key-mixing step of the 4th round. (We can do this because we only need to invert the two left-most S-boxes of the 4th round in order to derive those 8 bits.) When we guess the correct value for the initial 8 bits of k_5, we expect the 8-bit differential 0110 0011 to occur with probability at least 1/64. Heuristically, an incorrect guess yields the expected differential only with probability $2^{-8} = 1/256$. By setting L large enough, we can (with high probability) identify the correct value.

Differential attacks in practice. Differential cryptanalysis is very powerful, and has been used to attack real ciphers. A prominent example is FEAL-8, which was proposed as an alternative to DES in 1987. A differential attack on FEAL-8 was found that requires just 1,000 chosen plaintexts. In 1991, it took less than 2 minutes using this attack to find the entire key. Today, any proposed cipher is tested for resistance to differential cryptanalysis.

A differential attack was the first attack on DES to require less time than a simple brute-force search. While an interesting theoretical result, the attack is not of significant concern in practice since it requires 2^{47} chosen plaintexts. It

[7] This is a lower bound on the probability of the differential, since there may be other differences in the intermediate values that result in the same difference in the outputs.

is very difficult for an attacker to obtain this many chosen plaintext/ciphertext pairs in most real-world applications. Interestingly, small modifications to the S-boxes of DES make the cipher much more vulnerable to differential attacks. Personal testimony of the DES designers (after differential attacks were discovered in the outside world) has confirmed that the S-boxes of DES were designed specifically to thwart differential attacks.

Linear cryptanalysis. Linear cryptanalysis was developed by Matsui in the early 1990s. We will only describe the technique at a high level. The basic idea is to consider linear relationships between the input and output that hold with higher probability than would be expected for a random function. In more detail, say that bit positions $i_1, \ldots, i_{in}$ and $i'_1, \ldots, i'_{out}$ have *linear bias* ε if, for uniform x and k, and $y \stackrel{\text{def}}{=} F_k(x)$, it holds that

$$\left| \Pr[x_{i_1} \oplus \cdots \oplus x_{i_{in}} \oplus y_{i'_1} \oplus \cdots \oplus y_{i'_{out}} = 0] - \frac{1}{2} \right| = \varepsilon,$$

where x_i, y_i denote the ith bits of x and y. For a random function and any fixed set of bit positions, we expect the bias to be close to 0. Matsui showed how to use a large enough bias in a cipher F to find the secret key. Besides giving another method for attacking ciphers, an important feature of this attack is that it does not require *chosen* plaintexts, but rather *known* plaintexts suffice. This is very significant, since an encrypted file can provide a huge amount of known plaintext, whereas gathering encryptions of chosen plaintexts is much more difficult. Matsui showed that DES can be broken with just 2^{43} known plaintext/ciphertext pairs.

Impact on block-cipher design. Modern block ciphers are designed and evaluated based, in part, on their resistance to differential and linear cryptanalysis. When constructing a block cipher, designers choose S-boxes and other components so as to minimize differential probabilities and linear biases. We remark that it is not possible to eliminate *all* high-probability differentials in an S-box: any S-box will have *some* differential that occurs more frequently than others. Still, these deviations can be minimized. Moreover, increasing the number of rounds (and choosing the mixing permutation carefully) can both reduce the differential probabilities as well as make it more difficult for cryptanalysts to find any differentials to exploit.

6.3 Hash Functions

Recall from Chapter 5 that the primary security requirement for a hash function H is *collision resistance*; that is, it should be difficult to find a collision, or distinct inputs x, x' such that $H(x) = H(x')$. (We drop mention

of any key here, since real-world hash functions are generally unkeyed.) If the hash function has ℓ-bit output length, then the best we can hope for is that it should be infeasible to find a collision using substantially fewer than $2^{\ell/2}$ invocations of H. (See Section 5.4.1.) We would also like the hash function to achieve (second) preimage resistance against attacks running in time much less than 2^{ℓ}, although we do not consider such attacks in our discussion here.

Hash functions are generally constructed in two steps. First, a *compression function* (i.e., a fixed-length hash function) h is designed; next, some mechanism is used to extend h so as to handle arbitrary input lengths. In Section 5.2 we have already shown one approach—the Merkle–Damgård transform—for the second step. Here, we explore a technique for designing the underlying compression function. We also discuss some hash functions used in practice. A theoretical construction of a compression function based on a number-theoretic assumption is given in Section 8.4.2.

6.3.1 Hash Functions from Block Ciphers

Perhaps surprisingly, it is possible to build a collision-resistant compression function from a block cipher that satisfies certain additional properties. There are several ways to do this; one of the most common is via the *Davies–Meyer construction*. Let F be a block cipher with n-bit key length and ℓ-bit block length. We can then define the compression function $h : \{0,1\}^{n+\ell} \to \{0,1\}^{\ell}$ by $h(k,x) \stackrel{\text{def}}{=} F_k(x) \oplus x$. (See Figure 6.11.)

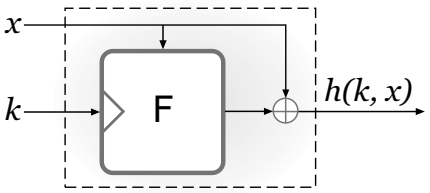


FIGURE 6.11: The Davies–Meyer construction.

We do not know how to prove collision resistance of the resulting compression function based only on the assumption that F is a strong pseudorandom permutation, and in fact there are reasons to believe such a proof is not possible. We *can*, however, prove collision resistance if we are willing to model F as an *ideal cipher*. The ideal-cipher model is a strengthening of the random-oracle model (see Section 5.5), in which we posit that all parties have access to an oracle for a random keyed permutation $F : \{0,1\}^n \times \{0,1\}^{\ell} \to \{0,1\}^{\ell}$ as well as its inverse F^{-1} (i.e., such that $F^{-1}(k, F(k,x)) = x$ for all k, x). Another way to think of this is that each key $k \in \{0,1\}^n$ specifies an independent, uniform permutation $F(k,\cdot)$ on ℓ-bit strings. As in the random-oracle model, the *only* way to compute F (or F^{-1}) is to explicitly query the oracle with (k,x) and receive back $F(k,x)$ (or $F^{-1}(k,x)$).

Analyzing constructions in the ideal-cipher model comes with all the advantages and disadvantages of working in the random-oracle model, as discussed at length in Section 5.5. We only add here that the ideal-cipher model implies the absence of *related-key attacks*, in the sense that (as we have just said) the permutations $F(k,\cdot)$ and $F(k',\cdot)$ must behave independently even if, for example, k and k' differ in only a single bit. In addition, there can be no "weak keys" k (say, the all-0 key) for which $F(k,\cdot)$ is easily distinguishable from random. It also means that $F(k,\cdot)$ should "behave randomly" *even when k is known*. For any real-world cipher F, these properties do *not* necessarily hold (and are not even well defined) even if F is a strong pseudorandom permutation, and the reader may note that we have not discussed these properties in any of our analysis of real-world block-cipher constructions. (In fact, DES and triple-DES do not satisfy these properties.) Any block cipher being used to instantiate an ideal cipher must be evaluated with respect to these more stringent requirements.

We prove the following theorem in a concrete setting, but the proof could be adapted easily for the asymptotic setting as well.

THEOREM 6.5 *If F is modeled as an ideal cipher, then the Davies–Meyer construction yields a collision-resistant compression function. Concretely, any attacker making $q < 2^{\ell/2}$ queries to its ideal-cipher oracles finds a collision with probability at most $q^2/2^\ell$.*

PROOF To be clear, we consider here the probabilistic experiment in which F is sampled at random (more precisely, for each $k \in \{0,1\}^n$ the function $F(k,\cdot) : \{0,1\}^\ell \to \{0,1\}^\ell$ is chosen uniformly from the set Perm_ℓ of permutations on ℓ-bit strings) and then the attacker is given oracle access to F and F^{-1}. The attacker then tries to find a colliding pair (k,x), (k',x'), i.e., for which $F(k,x)\oplus x = F(k',x')\oplus x'$. No computational bounds are placed on the attacker other than bounding the number of oracle queries it makes. We assume that if the attacker outputs a colliding pair $(k,x),(k',x')$ then it has previously made the oracle queries necessary to compute the values $F(k,x)$ and $F(k',x')$. We also assume the attacker never makes the same query more than once, and never queries $F^{-1}(k,y)$ once it has learned that $y = F(k,x)$ (and vice versa). All these assumptions are without loss of generality.

Consider the ith query the attacker makes to its oracles. A query (k_i,x_i) to F reveals only the hash value $h_i \stackrel{\text{def}}{=} h(k_i,x_i) = F(k_i,x_i)\oplus x_i$; similarly, a query to F^{-1} giving the result $x_i = F^{-1}(k_i,y_i)$ yields only the hash value $h_i \stackrel{\text{def}}{=} h(k_i,x_i) = y_i \oplus F^{-1}(k_i,y_i)$. The attacker does not obtain a collision unless $h_i = h_j$ for some $i \neq j$.

Fix i,j with $i > j$ and consider the probability that $h_i = h_j$. At the time of the ith query, the value of h_j is fixed. A collision between h_i and h_j is obtained on the ith query only if the attacker queries (k_i,x_i) to F and obtains the result $F(k_i,x_i) = h_j \oplus x_i$, or queries (k_i,y_i) to F^{-1} and obtains

the result $F^{-1}(k_i, y_i) = h_j \oplus y_i$. Either event occurs with probability at most $1/(2^\ell - (i-1))$ since, for example, $F(k_i, x_i)$ is uniform over $\{0,1\}^\ell$ except that it cannot be equal to any value $F(k_i, x)$ already defined by the attacker's (at most) $i-1$ previous oracle queries using key k_i. Since $i \leq q < 2^{\ell/2}$, the probability that $h_i = h_j$ is at most $2/2^\ell$.

Taking a union bound over all $\binom{q}{2} < q^2/2$ distinct pairs i, j gives the result stated in the theorem. ■

Davies–Meyer and DES. As we have mentioned above, one must take care when instantiating the Davies–Meyer construction with any concrete block cipher, since the cipher must satisfy additional properties (beyond being a strong pseudorandom permutation) in order for the resulting construction to be secure. In Exercise 6.21 we explore what goes wrong when DES is used in the Davies–Meyer construction.

This should serve as a *warning* that the proof of security for the Davies–Meyer construction in the ideal-cipher model does not necessarily translate into real security when instantiated with a real cipher. Nevertheless, as we will describe below, this paradigm has been used to construct practical hash functions that have resisted attack (but specifically when the block cipher at the center of the construction was designed specifically for this purpose).

In conclusion, the Davies–Meyer construction is a useful paradigm for constructing collision-resistant compression functions. However, it should *not* be applied to block ciphers designed for encryption, like DES and AES.

6.3.2 MD5

MD5 is a hash function with a 128-bit output length. It was designed in 1991 and for some time was believed to be collision resistant. Over a period of several years, various weaknesses began to be found in MD5 but these did not appear to lead to any easy way to find collisions. Shockingly, in 2004 a team of Chinese cryptanalysts presented a new method for finding collisions in MD5; they were easily able to convince others that their approach was correct by demonstrating an explicit collision! Since then, the attack has been improved and today collisions can be found in under a minute on a desktop PC. In addition, the attacks have been extended so that even "controlled collisions" (e.g., two postscript files generating arbitrary viewable content) can be found.

Due to these attacks, MD5 should not be used anywhere cryptographic security is needed. We mention MD5 only because it is still found in legacy code.

6.3.3 SHA-0, SHA-1, and SHA-2

The *Secure Hash Algorithm* (SHA) refers to a series of cryptographic hash functions standardized by NIST. Perhaps the most well known of these is SHA-1, which was introduced in 1995. This algorithm has a 160-bit output length and supplanted a predecessor called SHA-0, which was withdrawn due to unspecified flaws discovered in that algorithm.

At the time of this writing, an explicit collision has yet to be found in SHA-1. However, theoretical analysis over the past few years indicates that collisions in SHA-1 can be found using significantly fewer than the 2^{80} hash-function evaluations that would be necessary using a birthday attack, and it is conjectured that a collision will be found soon. It is therefore recommended to migrate to SHA-2, which does not currently appear to have the same weaknesses. SHA-2 is comprised of two related functions: SHA-256 and SHA-512, with 256- and 512-bit output lengths, respectively.

All hash functions in the SHA family are constructed using the same basic design, which incorporates components we have already seen: A compression function is first defined by applying the Davies–Meyer construction to a block cipher, and this is then extended to support arbitrary length inputs using the Merkle–Damgård transform. One interesting thing here is that the block cipher in each case was designed specifically for building the compression function. In fact, it was only retroactively that the underlying components in the compression functions were isolated and analyzed as the block ciphers SHACAL-1 (for SHA-1) and SHACAL-2 (for SHA-2). These ciphers are themselves intriguing, as they have large block lengths (160 and 256 bits, respectively) and 512-bit keys.

6.3.4 SHA-3 (Keccak)

In the aftermath of the collision attack on MD5 and the theoretical weaknesses found in SHA-1, NIST announced in late 2007 a public competition to design a new cryptographic hash function to be called SHA-3. Submitted algorithms were required to support at least 256- and 512-bit output lengths. As in the case of the AES competition from roughly 10 years earlier, the competition was completely open and transparent; anyone could submit an algorithm for consideration, and the public was invited to submit their opinions on any of the candidates. The 51 first-round candidates were narrowed down to 14 in December 2008, and these were further reduced to five finalists in 2010. The remaining candidates were subject to intense scrutiny by the cryptographic community over the next two years. In October 2012, NIST announced the selection of *Keccak* as the winner of the competition. As of the time of this writing, this algorithm is undergoing standardization as the next-generation replacement for SHA-2.

Keccak is unusual in several respects. (Interestingly, one of the reasons Keccak was chosen is because its structure is very different from that of SHA-1 and SHA-2.) At its core, it is based on an *unkeyed* permutation f with a large block length of 1600 bits; this is radically different from, e.g., the Davies–Meyer construction, which relies on a *keyed* permutation. Furthermore, Keccak does not use the Merkle–Damgård transform to handle arbitrary input lengths. Instead, it uses a newer approach called the *sponge construction*. Keccak—and the sponge construction more generally—can be analyzed in the *random-permutation model* in which we postulate that parties

have access to an oracle for a random permutation $f : \{0,1\}^\ell \to \{0,1\}^\ell$ (and possibly its inverse). This is weaker than the ideal-cipher model; indeed, we can easily obtain a random permutation in the ideal-cipher model by simply fixing the key to the cipher to be any constant value.

It will be fascinating to watch the new hash standard evolve, and to see how quickly developers adapt from SHA-1/SHA-2 to the newer SHA-3.

References and Additional Reading

Additional information on LFSRs and stream ciphers can be found in the *Handbook of Applied Cryptography* [120] or the more recent text by Paar and Pelzl [135]. Further details regarding eSTREAM, as well as a detailed specification of Trivium, can be found at `http://www.ecrypt.eu.org/stream`. See the work of AlFardan et al. [9] for a recent survey of attacks on RC4.

The confusion-diffusion paradigm and substitution-permutation networks were introduced by Shannon [154] and Feistel [64]. See the thesis of Heys [90] for further information regarding SPN design. Better generic attacks on three-round SPNs than what we have shown here are known [31]. Miles and Viola [126] give a theoretical analysis of SPNs.

Feistel networks were first described in [64]. A theoretical analysis of Feistel networks was given by Luby and Rackoff [116]; see Chapter 7.

More details on DES, AES, and block-cipher constructions in general can be found in the text by Knudsen and Robshaw [106]. The meet-in-the-middle attack on double encryption is due to Diffie and Hellman [59]. The attack on two-key triple encryption mentioned in the text (and explored in Exercise 6.13) is by Merkle and Hellman [124]. Theoretical analysis of the security of double and triple encryption can be found in [6, 24].

DESX is another technique for increasing the effective key length of DES. The secret key consists of values $k_i, k_o \in \{0,1\}^{64}$, and $k \in \{0,1\}^{56}$, and the cipher is defined by

$$DESX_{k_i,k,k_o}(x) \stackrel{\text{def}}{=} k_o \oplus DES_k(x \oplus k_i).$$

This methodology was first studied by Even and Mansour [63] in a slightly different context. Its application to DES was proposed in unpublished work by Rivest, and its security was later analyzed by Kilian and Rogaway [105, 149].

Differential cryptanalysis was introduced by Biham and Shamir [29] and its application to DES is described in a book by those authors [30]. Coppersmith [45] describes design principles of the DES S-boxes in light of the public discovery of differential cryptanalysis. Linear cryptanalysis was discovered by Matsui [118], who shows its application to DES there. For more information on these advanced cryptanalytic techniques, we refer the reader to the tutorial

on differential and linear cryptanalysis by Heys [91] or to the aforementioned book by Knudsen and Robshaw [106].

For further information about MD5 and SHA-1 see [120]. Note, however, that their treatment pre-dates the attacks by Wang et al. [175, 174]. Constructions of compression functions from block ciphers are analyzed in [143, 33]. The sponge construction is described and analyzed by Bertoni et al. [28]. For additional details about the SHA-3 competition, see the NIST webpage at `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

Exercises

6.1 Assume a degree-6 LFSR with $c_0 = c_5 = 1$ and $c_1 = c_2 = c_3 = c_4 = 0$.

(a) What are the first 10 bits output by this LFSR if it starts in initial state $(1, 1, 1, 1, 1, 1)$?

(b) Is this LFSR maximal length?

6.2 In this question we consider a nonlinear combination generator, where we have a degree-n LFSR but the output at each time step is not s_0 but instead $g(s_0, \ldots, s_{n-1})$ for some nonlinear function g. Assume the feedback coefficients of the LFSR are known, but its initial state is not. Show that each of the following choices of g does not yield a good pseudorandom generator:

(a) $g(s_0, \ldots, s_{n-1}) = s_0 \wedge s_1$.

(b) $g(s_0, \ldots, s_{n-1}) = (s_0 \wedge s_1) \oplus s_2$.

6.3 Let F be a block cipher with n-bit key length and block length. Say there is a key-recovery attack on F that succeeds with probability 1 using n chosen plaintexts and minimal computational effort. Prove formally that F cannot be a pseudorandom permutation.

6.4 In our attack on a one-round SPN, we considered a block length of 64 bits and 16 S-boxes that each take a 4-bit input. Repeat the analysis for the case of 8 S-boxes, each taking an 8-bit input. What is the complexity of the attack now? Repeat the analysis again with a 128-bit block length and 16 S-boxes that each take an 8-bit input.

6.5 Consider a modified SPN where instead of carrying out the key-mixing, substitution, and permutation steps in alternating order for r (full) rounds, the cipher instead first applies r rounds of key mixing, then carries out r rounds of substitution, and finally applies r mixing permutations. Analyze the security of this construction.

6.6 In this question we assume a two-round SPN with 64-bit block length.

(a) Assume independent 64-bit sub-keys are used in each round, so the master key is 192 bits long. Show a key-recovery attack using much less than 2^{192} time.

(b) Assume the first and third sub-keys are equal, and the second sub-key is independent, so the master key is 128 bits long. Show a key-recovery attack using much less than 2^{128} time.

6.7 What is the output of an r-round Feistel network when the input is (L_0, R_0) in each of the following two cases:

(a) Each round function outputs all 0s, regardless of the input.

(b) Each round function is the identity function.

6.8 Let $\mathsf{Feistel}_{f_1,f_2}(\cdot)$ denote a two-round Feistel network using functions f_1 and f_2 (in that order). Show that if $\mathsf{Feistel}_{f_1,f_2}(L_0, R_0) = (L_2, R_2)$, then $\mathsf{Feistel}_{f_2,f_1}(R_2, L_2) = (R_0, L_0)$.

6.9 For this exercise, rely on the description of DES given in this chapter, but use the fact that in the actual construction of DES the two halves of the output of the final round of the Feistel network are swapped. That is, if the output of the final round of the Feistel network is (L_{16}, R_{16}), then the output of DES is (R_{16}, L_{16}).

(a) Show that the only difference between computation of DES_k and DES_k^{-1} is the order in which sub-keys are used. (Rely on the previous exercise.)

(b) Show that for $k = 0^{56}$ it holds that $DES_k(DES_k(x)) = x$.

Hint: Consider the sub-keys generated from this key.

(c) Find three other DES keys with the same property. These keys are known as *weak keys* for DES. (Note: the keys you find will differ from the actual weak keys of DES because of differences in our presentation.)

(d) Do these 4 weak keys represent a serious vulnerability in the use of triple-DES as a pseudorandom permutation? Explain.

6.10 Show that DES has the property that $DES_k(x) = \overline{DES_{\bar{k}}(\bar{x})}$ for every key k and input x (where $\bar{z}$ denotes the bitwise complement of z). (This is called the *complementarity property* of DES.) Does this represent a serious vulnerability in the use of triple-DES as a pseudorandom permutation? Explain.

6.11 Describe attacks on the following modifications to DES:

(a) Each sub-key is 32 bits long, and the round function simply XORs the sub-key with the input to the round (i.e., $\hat{f}(k, R) = k_i \oplus R$). For this question, the key schedule is unimportant and you can treat the sub-keys k_i as independent keys.

(b) Instead of using different sub-keys in every round, the same 48-bit sub-key is used in every round. Show how to distinguish the cipher from a random permutation in $\ll 2^{48}$ time.

Hint: Exercises 6.8 and 6.9 may help...

6.12 (This exercise relies on Exercise 6.9.) Our goal is to show that for any weak key k of DES, it is easy to find an input x such that $DES_k(x) = x$.

(a) Assume we evaluate DES_k on input (L_0, R_0), and the output after 8 rounds of the Feistel network is (L_8, R_8) with $L_8 = R_8$. Show that the output of $DES_k(L_0, R_0)$ is (L_0, R_0). (Recall from Exercise 6.9 that DES swaps the two halves of the 16th round of the Feistel network before outputting the result.)

(b) Show how to find an input (L_0, R_0) with the property in part (a).

6.13 This question illustrates an attack on two-key triple encryption. Let F be a block cipher with n-bit block length and key length, and set $F'_{k_1,k_2}(x) \stackrel{\text{def}}{=} F_{k_1}(F^{-1}_{k_2}(F_{k_1}(x)))$.

(a) Assume that given a pair (m_1, m_2) it is possible to find in *constant* time all keys k_2 such that $m_2 = F^{-1}_{k_2}(m_1)$. Show how to recover the entire key for F' (with high probability) in time roughly 2^n using three known input/output pairs.

(b) In general, it will *not* be possible to find k_2 as above in constant time. However, show that by using a preprocessing step taking 2^n time it is possible, given m_2, to find in (essentially) constant time all keys k_2 such that $m_2 = F^{-1}_{k_2}(0^n)$.

(c) Assume k_1 is known and that the pre-processing step above has already been run. Show how to use the value $y = F'_{k_1,k_2}(x)$ for a single *chosen* plaintext x to determine k_2 in constant time.

(d) Put the above components together to devise an attack that recovers the entire key of F' by running in roughly 2^n time and requesting the encryption of roughly 2^n chosen inputs.

6.14 Say the key schedule of DES is modified as follows: the left half of the master key is used to derive all the sub-keys in rounds 1–8, while the right half of the master key is used to derive all the sub-keys in rounds 9–16. Show an attack on this modified scheme that recovers the entire key in time roughly 2^{28}.

6.15 Let $f : \{0,1\}^m \times \{0,1\}^\ell \to \{0,1\}^\ell$ and $g : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^\ell$ be secure block ciphers with $m > n$, and define $F_{k_1,k_2}(x) = f_{k_1}(g_{k_2}(x))$. Show a key-recovery attack on F using time $\mathcal{O}(2^m)$ and space $\mathcal{O}(\ell \cdot 2^n)$.

6.16 Define $DESY_{k,k'}(x) = DES_k(x \oplus k')$. The key length of $DESY$ is 120 bits. Show a key-recovery attack on $DESY$ taking time and space $\approx 2^{64}$.

6.17 Choose random S-boxes and mixing permutations for SPNs of different sizes, and develop differential attacks against them. We recommend trying five-round SPNs with 16-bit and 24-bit block lengths, using S-boxes with 4-bit input/output. Write code to compute the differential tables, and to carry out the attack.

6.18 Implement the time/space tradeoff for 40-bit DES (i.e., fix the first 16 bits of the key of DES to 0). Calculate the time and memory needed, and empirically estimate the probability of success. Experimentally verify the increase in success probability as the number of tables is increased. (Warning: this is a big project!)

6.19 For each of the following constructions of a compression function h from a block cipher F, either show an attack or prove collision resistance in the ideal-cipher model:

(a) $h(k,x) = F_k(x)$.

(b) $h(k,x) = F_k(x) \oplus k \oplus x$.

(c) $h(k,x) = F_k(x) \oplus k$.

6.20 Consider using DES to construct a compression function in the following way: Define $h : \{0,1\}^{112} \to \{0,1\}^{64}$ as $h(x_1,x_2) \stackrel{\text{def}}{=} DES_{x_1}(DES_{x_2}(0^{64}))$ where $|x_1| = |x_2| = 56$.

(a) Write down an explicit collision in h.
Hint: Use Exercise 6.9(a–b).

(b) Show how to find a preimage of an arbitrary value y (that is, x_1, x_2 such that $h(x_1 \| x_2) = y$) in roughly 2^{56} time.

(c) Show a more clever preimage attack that runs in roughly 2^{32} time and succeeds with high probability.
Hint: Rely on the results of Appendix A.4.

6.21 Let F be a block cipher for which it is easy to find fixed points for some key: namely, there is a key k for which it is easy to find inputs x for which $F_k(x) = x$. Find a collision in the Davies–Meyer construction when applied to F. (Consider this in light of Exercise 6.12.)

Chapter 7

*Theoretical Constructions of Symmetric-Key Primitives

In Chapter 3 we introduced the notion of pseudorandomness and defined some basic cryptographic primitives including pseudorandom generators, functions, and permutations. We showed in Chapters 3 and 4 that these primitives serve as the building blocks for all of private-key cryptography. As such, it is of great importance to understand these primitives from a theoretical point of view. In this chapter we formally introduce the concept of *one-way functions*—functions that are, informally, easy to compute but hard to invert—and show how pseudorandom generators, functions, and permutations can be constructed under the sole assumption that one-way functions exist.[1] Moreover, we will see that one-way functions are necessary for "non-trivial" private-key cryptography. That is: *the existence of one-way functions is equivalent to the existence of all (non-trivial) private-key cryptography.* This is one of the major contributions of modern cryptography.

The constructions we show in this chapter should be viewed as complementary to the constructions of stream ciphers and block ciphers discussed in the previous chapter. The focus of the previous chapter was on how various cryptographic primitives are currently realized in practice, and the intent of that chapter was to introduce some basic approaches and design principles that are used. Somewhat disappointing, though, was the fact that none of the constructions we showed could be *proven* secure based on any weaker (i.e., more reasonable) assumptions. In contrast, in the present chapter we will prove that it is possible to construct pseudorandom permutations starting from the very mild assumption that one-way functions exist. This assumption is more palatable than assuming, say, that AES is a pseudorandom permutation, both because it is a qualitatively weaker assumption and also because we have a number of candidate, number-theoretic one-way functions that have been studied for many years, even before the advent of cryptography. (See the very beginning of Chapter 6 for further discussion of this point.) The downside, however, is that the constructions we show here are all far less efficient than those of Chapter 6, and thus are not actually used. It remains an important challenge for cryptographers to "bridge this gap" and develop provably

[1] This is not quite true since we are for the most part going to rely on one-way *permutations* in this chapter. But it is known that one-way functions suffice.

secure constructions of pseudorandom generators, functions, and permutations whose efficiency is comparable to the best available stream ciphers and block ciphers.

Collision-resistant hash functions. In contrast to the previous chapter, here we do not consider collision-resistant hash functions. The reason is that constructions of such hash functions from one-way functions are unknown and, in fact, there is evidence suggesting that such constructions are impossible. We will turn to provable constructions of collision-resistant hash functions—based on specific, number-theoretic assumptions—in Section 8.4.2.

A note regarding this chapter. The material in this chapter is somewhat more advanced than the material in the rest of this book. This material is not used explicitly elsewhere, and so this chapter can be skipped if desired. Having said this, we have tried to present the material in such a way that it is understandable (with effort) to an advanced undergraduate or beginning graduate student. We encourage all readers to peruse Sections 7.1 and 7.2, which introduce one-way functions and provide an overview of the rest of this chapter. We believe that familiarity with at least some of the topics covered here is important enough to warrant the effort.

7.1 One-Way Functions

In this section we formally define one-way functions, and then briefly discuss some candidates that are widely believed to satisfy this definition. (We will see more examples of conjectured one-way functions in Chapter 8.) We next introduce the notion of *hard-core predicates*, which can be viewed as encapsulating the hardness of inverting a one-way function and will be used extensively in the constructions that follow in subsequent sections.

7.1.1 Definitions

A one-way function $f : \{0,1\}^* \to \{0,1\}^*$ is easy to compute, yet hard to invert. The first condition is easy to formalize: we will simply require that f be computable in polynomial time. Since we are ultimately interested in building cryptographic schemes that are hard for a probabilistic polynomial-time adversary to break except with negligible probability, we will formalize the second condition by requiring that it be infeasible for any probabilistic polynomial-time algorithm to invert f—that is, to find a preimage of a given value y—except with negligible probability. A technical point is that this probability is taken over an experiment in which y is generated by choosing a uniform element x of the domain of f and then setting $y := f(x)$ (rather than

choosing y uniformly from the range of f). The reason for this should become clear from the constructions we will see in the remainder of the chapter.

Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. Consider the following experiment defined for any algorithm $\mathcal{A}$ and any value n for the security parameter:

The inverting experiment $\mathsf{Invert}_{\mathcal{A},f}(n)$

1. *Choose uniform* $x \in \{0,1\}^n$*, and compute* $y := f(x)$*.*
2. $\mathcal{A}$ *is given* 1^n *and* y *as input, and outputs* x'*.*
3. *The output of the experiment is defined to be* 1 *if* $f(x') = y$*, and* 0 *otherwise.*

We stress that $\mathcal{A}$ need not find the original preimage x; it suffices for $\mathcal{A}$ to find any value x' for which $f(x') = y = f(x)$. We give the security parameter 1^n to $\mathcal{A}$ in the second step to stress that $\mathcal{A}$ may run in time polynomial in the security parameter n, regardless of the length of y.

We can now define what it means for a function f to be one-way.

DEFINITION 7.1 *A function* $f : \{0,1\}^* \to \{0,1\}^*$ *is* one-way *if the following two conditions hold:*

1. **(Easy to compute:)** *There exists a polynomial-time algorithm* M_f *computing* f*; that is,* $M_f(x) = f(x)$ *for all* x*.*
2. **(Hard to invert:)** *For every probabilistic polynomial-time algorithm* $\mathcal{A}$*, there is a negligible function* negl *such that*

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

Notation. In this chapter we will often make the probability space more explicit by subscripting (part of) it in the probability notation. For example, we can succinctly express the second requirement in the definition above as follows: For every probabilistic polynomial-time algorithm $\mathcal{A}$, there exists a negligible function negl such that

$$\Pr_{x \leftarrow \{0,1\}^n}\left[\mathcal{A}(1^n, f(x)) \in f^{-1}(f(x))\right] \leq \mathsf{negl}(n).$$

(Recall that $x \leftarrow \{0,1\}^n$ means that x is chosen uniformly from $\{0,1\}^n$.) The probability above is also taken over the randomness used by $\mathcal{A}$, which here is left implicit.

Successful inversion of one-way functions. A function that is *not* one-way is not necessarily easy to invert all the time (or even "often"). Rather, the converse of the second condition of Definition 7.1 is that there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a non-negligible function γ

such that $\mathcal{A}$ inverts $f(x)$ with probability at least $\gamma(n)$ (where the probability is taken over uniform choice of $x \in \{0,1\}^n$ and the randomness of $\mathcal{A}$). This means, in turn, that there exists a positive polynomial $p(\cdot)$ such that for *infinitely many values of* n, algorithm $\mathcal{A}$ inverts f with probability at least $1/p(n)$. Thus, if there exists an $\mathcal{A}$ that inverts f with probability n^{-10} for all even values of n (but always fails to invert f when n is odd), then f is not one-way—even though $\mathcal{A}$ only succeeds on half the values of n, and only succeeds with probability n^{-10} (for values of n where it succeeds at all).

Exponential-time inversion. Any one-way function can be inverted at any point y in exponential time, by simply trying all values $x \in \{0,1\}^n$ until a value x is found such that $f(x) = y$. Thus, the existence of one-way functions is inherently an assumption about *computational complexity* and *computational hardness*. That is, it concerns a problem that can be solved in principle but is assumed to be hard to solve efficiently.

One-way permutations. We will often be interested in one-way functions with additional structural properties. We say a function f is *length-preserving* if $|f(x)| = |x|$ for all x. A one-way function that is length-preserving and one-to-one is called a *one-way permutation*. If f is a one-way permutation, then any value y has a unique preimage $x = f^{-1}(y)$. Nevertheless, it is still hard to find x in polynomial time.

One-way function/permutation families. The above definitions of one-way functions and permutations are convenient in that they consider a single function over an infinite domain and range. However, most candidate one-way functions and permutations do not fit neatly into this framework. Instead, there is an algorithm that generates some set of parameters I which define a function f_I; one-wayness here means essentially that f_I should be one-way with all but negligible probability over choice of I. Because each value of I defines a different function, we now refer to *families* of one-way functions (resp., permutations). We give the definition now, and refer the reader to the next section for a concrete example. (See also Section 8.4.1.)

DEFINITION 7.2 *A tuple* $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ *of probabilistic polynomial-time algorithms is a* function family *if the following hold:*

1. *The* parameter-generation algorithm Gen*, on input* 1^n*, outputs parameters* I *with* $|I| \geq n$*. Each value of* I *output by* Gen *defines sets* $\mathcal{D}_I$ *and* $\mathcal{R}_I$ *that constitute the domain and range, respectively, of a function* f_I*.*

2. *The* sampling algorithm Samp*, on input* I*, outputs a uniformly distributed element of* $\mathcal{D}_I$*.*

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* permutation family *if for each value of* I *output by* $\mathsf{Gen}(1^n)$*, it holds that* $\mathcal{D}_I = \mathcal{R}_I$ *and the function* $f_I : \mathcal{D}_I \to \mathcal{D}_I$ *is a bijection.*

Let Π be a function family. What follows is the natural analogue of the experiment introduced previously.

The inverting experiment $\mathsf{Invert}_{\mathcal{A},\Pi}(n)$**:**

1. $\mathsf{Gen}(1^n)$ *is run to obtain* I*, and then* $\mathsf{Samp}(I)$ *is run to obtain a uniform* $x \in \mathcal{D}_I$*. 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* 1 *if* $f_I(x') = y$*.*

DEFINITION 7.3 *A function/permutation family* $\Pi = (\mathsf{Gen}, \mathsf{Samp}, f)$ *is* one-way *if for all probabilistic polynomial-time algorithms* $\mathcal{A}$ *there exists a negligible function* negl *such that*

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

Throughout this chapter we work with one-way functions/permutations over an infinite domain (as in Definition 7.1), rather than working with families of one-way functions/permutations. This is primarily for convenience, and does not significantly affect any of the results. (See Exercise 7.7.)

7.1.2 Candidate One-Way Functions

One-way functions are of interest only if they exist. We do not know how to prove they exist unconditionally (this would be a major breakthrough in complexity theory), so we must conjecture or assume their existence. Such a conjecture is based on the fact that several natural computational problems have received much attention, yet still have no polynomial-time algorithm for solving them. Perhaps the most famous such problem is *integer factorization*, i.e., finding the prime factors of a large integer. It is easy to multiply two numbers and obtain their product, but difficult to take a number and find its factors. This leads us to define the function $f_{\text{mult}}(x, y) = x \cdot y$. If we do not place any restriction on the lengths of x and y, then f_{mult} is easy to invert: with high probability $x \cdot y$ will be *even*, in which case $(2, xy/2)$ is an inverse. This issue can be addressed by restricting the domain of f_{mult} to equal-length *primes* x and y. We return to this idea in Section 8.2.

Another candidate one-way function, not relying directly on number theory, is based on the *subset-sum problem* and is defined by

$$f_{\text{ss}}(x_1, \ldots, x_n, J) = \left(x_1, \ldots, x_n, \left[\textstyle\sum_{j \in J} x_j \bmod 2^n\right]\right),$$

where each x_i is an n-bit string interpreted as an integer, and J is an n-bit string interpreted as specifying a subset of $\{1, \ldots, n\}$. Inverting f_{ss} on an output $(x_1, \ldots, x_n, y)$ requires finding a subset $J' \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in J'} x_j = y \bmod 2^n$. Students who have studied $\mathcal{NP}$-completeness may recall that this problem is $\mathcal{NP}$-complete. But even $\mathcal{P} \neq \mathcal{NP}$ does not imply that f_{ss} is one-way: $\mathcal{P} \neq \mathcal{NP}$ would mean that every polynomial-time algorithm fails to solve the subset-sum problem on *at least one* input, whereas for f_{ss} to be a one-way function it is required that every polynomial-time algorithm fails to solve the subset-sum problem (at least for certain parameters) *almost always*. Thus, our belief that the function above is one-way is based on the lack of known algorithms to solve this problem even with "small" probability on random inputs, and not merely on the fact that the problem is $\mathcal{NP}$-complete.

We conclude by showing a family of *permutations* that is believed to be one-way. Let Gen be a probabilistic polynomial-time algorithm that, on input 1^n, outputs an n-bit prime p along with a special element $g \in \{2, \ldots, p-1\}$. (The element g should be a *generator* of $\mathbb{Z}_p^*$; see Section 8.3.3.) Let Samp be an algorithm that, given p and g, outputs a uniform integer $x \in \{1, \ldots, p-1\}$. Finally, define

$$f_{p,g}(x) = [g^x \bmod p].$$

(The fact that $f_{p,g}$ can be computed efficiently follows from the results in Appendix B.2.3.) It can be shown that this function is one-to-one, and thus a permutation. The presumed difficulty of inverting this function is based on the conjectured hardness of the *discrete-logarithm problem*; we will have much more to say about this in Section 8.3.

Finally, we remark that very efficient one-way functions can be obtained from practical cryptographic constructions such as SHA-1 or AES under the assumption that they are collision resistant or a pseudorandom permutation, respectively; see Exercises 7.4 and 7.5. (Technically speaking, they cannot satisfy the definition of one-wayness since they have fixed-length input/output and so we cannot look at their asymptotic behavior. Nevertheless, it is plausible to conjecture that they are one-way in a concrete sense.)

7.1.3 Hard-Core Predicates

By definition, a one-way function is hard to invert. Stated differently: given $y = f(x)$, the value x cannot be computed *in its entirety* by any polynomial-time algorithm (except with negligible probability; we ignore this here). One might get the impression that nothing about x can be determined from $f(x)$ in polynomial time. This is *not* necessarily the case. Indeed, it is possible for $f(x)$ to "leak" a lot of information about x even if f is one-way. For a trivial example, let g be a one-way function and define $f(x_1, x_2) \stackrel{\text{def}}{=} (x_1, g(x_2))$, where $|x_1| = |x_2|$. It is easy to show that f is also a one-way function (this is left as an exercise), even though it reveals half its input.

For our applications, we will need to identify a specific piece of information about x that is "hidden" by $f(x)$. This motivates the notion of a *hard-core predicate*. A hard-core predicate $\mathsf{hc} : \{0,1\}^* \to \{0,1\}$ of a function f has the property that $\mathsf{hc}(x)$ is hard to compute with probability significantly better than $1/2$ given $f(x)$. (Since hc is a boolean function, it is always possible to compute $\mathsf{hc}(x)$ with probability $1/2$ by random guessing.) Formally:

DEFINITION 7.4 *A function* $\mathsf{hc} : \{0,1\}^* \to \{0,1\}$ *is a* hard-core predicate of a function f *if* hc *can be computed in polynomial time, and for every probabilistic polynomial-time algorithm* $\mathcal{A}$ *there is a negligible function* negl *such that*

$$\Pr_{x \leftarrow \{0,1\}^n}[\mathcal{A}(1^n, f(x)) = \mathsf{hc}(x)] \leq \frac{1}{2} + \mathsf{negl}(n),$$

where the probability is taken over the uniform choice of x *in* $\{0,1\}^n$ *and the randomness of* $\mathcal{A}$.

We stress that $\mathsf{hc}(x)$ is efficiently computable given x (since the function hc can be computed in polynomial time); the definition requires that $\mathsf{hc}(x)$ is hard to compute given $f(x)$. The above definition does not require f to be one-way; if f is a permutation, however, then it cannot have a hard-core predicate unless it is one-way. (See Exercise 7.13.)

Simple ideas don't work. Consider the predicate $\mathsf{hc}(x) \stackrel{\text{def}}{=} \bigoplus_{i=1}^n x_i$ where $x_1, \ldots, x_n$ denote the bits of x. One might hope that this is a hard-core predicate of any one-way function f: if f cannot be inverted, then $f(x)$ must hide at least one of the bits x_i of its preimage x, which would seem to imply that the exclusive-or of all of the bits of x is hard to compute. Despite its appeal, this argument is incorrect. To see this, let g be a one-way function and define $f(x) \stackrel{\text{def}}{=} (g(x), \bigoplus_{i=1}^n x_i)$. It is not hard to show that f is one-way. However, it is clear that $f(x)$ 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 a hard-core predicate of f. Extending this, one can show that for any fixed predicate hc, there is a one-way function f for which hc is not a hard-core predicate of f.

Trivial hard-core predicates. Some functions have "trivial" hard-core predicates. For example, let f be the function that drops the last bit of its input (i.e., $f(x_1 \cdots x_n) = x_1 \cdots x_{n-1}$). It is hard to determine x_n given $f(x)$ since x_n is independent of the output; thus, $\mathsf{hc}(x) = x_n$ is a hard-core predicate of f. However, f is not one-way. When we use hard-core predicates for our constructions, it will become clear why trivial hard-core predicates of this sort are of no use.

7.2 From One-Way Functions to Pseudorandomness

The goal of this chapter is to show how to construct pseudorandom generators, functions, and permutations based on any one-way function/permutation. In this section, we give an overview of these constructions. Details are given in the sections that follow.

A hard-core predicate from any one-way function. The first step is to show that a hard-core predicate exists for any one-way function. Actually, it remains open whether this is true; we show something weaker that suffices for our purposes. Namely, we show that given a one-way function f we can construct a *different* one-way function g along with a hard-core predicate of g.

THEOREM 7.5 (Goldreich–Levin theorem) *Assume one-way functions (resp., permutations) exist. Then there exists a one-way function (resp., permutation) g and a hard-core predicate hc of g.*

Let f be a one-way function. Functions g and hc are constructed as follows: set $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, for $|x| = |r|$, and define

$$\mathsf{hc}(x,r) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{n} x_i \cdot r_i,$$

where x_i (resp., r_i) denotes the ith bit of x (resp., r). Notice that if r is uniform, then $\mathsf{hc}(x,r)$ outputs the exclusive-or of a *random subset* of the bits of x. (When $r_i = 1$ the bit x_i is included in the XOR, and otherwise it is not.) The Goldreich–Levin theorem essentially states that if f is a one-way function then $f(x)$ hides the exclusive-or of a *random subset* of the bits of x.

Pseudorandom generators from one-way permutations. The next step is to show how a hard-core predicate of a one-way *permutation* can be used to construct a pseudorandom generator. (It is known that a hard-core predicate of a one-way *function* suffices, but the proof is extremely complicated and well beyond the scope of this book.) Specifically, we show:

THEOREM 7.6 *Let f be a one-way permutation and let hc be a hard-core predicate of f. Then, $G(s) \stackrel{\text{def}}{=} f(s) \,\|\, \mathsf{hc}(s)$ is a pseudorandom generator with expansion factor $\ell(n) = n+1$.*

As intuition for why G as defined in the theorem constitutes a pseudorandom generator, note first that the initial n bits of the output of $G(s)$ (i.e., the bits of $f(s)$) are *truly* uniformly distributed when s is uniformly distributed, by virtue of the fact that f is a permutation. Next, the fact that hc is a hard-core predicate of f means that $\mathsf{hc}(s)$ "looks random"—i.e., is *pseudo-*

random—even given $f(s)$ (assuming again that s is uniform). Putting these observations together, we see that the entire output of G is pseudorandom.

Pseudorandom generators with arbitrary expansion. The existence of a pseudorandom generator that stretches its seed by even a single bit (as we have just seen) is already highly non-trivial. But for applications (e.g., for efficient encryption of large messages as in Section 3.3), we need a pseudorandom generator with much larger expansion. Fortunately, we can obtain any polynomial expansion factor we want:

THEOREM 7.7 *If there exists a pseudorandom generator with expansion factor $\ell(n) = n+1$, then for any polynomial* poly *there exists a pseudorandom generator with expansion factor* $\mathsf{poly}(n)$.

We conclude that pseudorandom generators with arbitrary (polynomial) expansion can be constructed from any one-way permutation.

Pseudorandom functions/permutations from pseudorandom generators. Pseudorandom generators suffice for constructing EAV-secure private-key encryption schemes. For achieving CPA-secure private-key encryption (not to mention message authentication codes), however, we relied on pseudorandom functions. The following result shows that the latter can be constructed from the former:

THEOREM 7.8 *If there exists a pseudorandom generator with expansion factor $\ell(n) = 2n$, then there exists a pseudorandom function.*

In fact, we can do even more:

THEOREM 7.9 *If there exists a pseudorandom function, then there exists a strong pseudorandom permutation.*

Combining all the above theorems, as well as the results of Chapters 3 and 4, we have the following corollaries:

COROLLARY 7.10 *Assuming the existence of one-way permutations, there exist pseudorandom generators with any polynomial expansion factor, pseudorandom functions, and strong pseudorandom permutations.*

COROLLARY 7.11 *Assuming the existence of one-way permutations, there exist CCA-secure private-key encryption schemes and secure message authentication codes.*

As noted earlier, it is possible to obtain all these results based solely on the existence of one-way *functions*.

7.3 Hard-Core Predicates from One-Way Functions

In this section, we prove Theorem 7.5 by showing the following:

THEOREM 7.12 *Let f be a one-way function and define g by $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, where $|x| = |r|$. Define $\mathsf{gl}(x,r) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{n} x_i \cdot r_i$, where $x = x_1 \cdots x_n$ and $r = r_1 \cdots r_n$. Then gl is a hard-core predicate of g.*

Due to the complexity of the proof, we prove three successively stronger results culminating in what is claimed in the theorem.

7.3.1 A Simple Case

We first show that if there exists a polynomial-time adversary $\mathcal{A}$ that *always* correctly computes $\mathsf{gl}(x,r)$ given $g(x,r) = (f(x),r)$, then it is possible to invert f in polynomial time. Given the assumption that f is a one-way function, it follows that no such adversary $\mathcal{A}$ exists.

PROPOSITION 7.13 *Let f and gl be as in Theorem 7.12. If there exists a polynomial-time algorithm $\mathcal{A}$ such that $\mathcal{A}(f(x),r) = \mathsf{gl}(x,r)$ for all n and all $x, r \in \{0,1\}^n$, then there exists a polynomial-time algorithm $\mathcal{A}'$ such that $\mathcal{A}'(1^n, f(x)) = x$ for all n and all $x \in \{0,1\}^n$.*

PROOF We construct $\mathcal{A}'$ as follows. $\mathcal{A}'(1^n, y)$ computes $x_i := \mathcal{A}(y, e^i)$ for $i = 1, \ldots, n$, where e^i denotes the n-bit string with 1 in the ith position and 0 everywhere else. Then $\mathcal{A}'$ outputs $x = x_1 \cdots x_n$. Clearly $\mathcal{A}'$ runs in polynomial time.

In the execution of $\mathcal{A}'(1^n, f(\hat{x}))$, the value x_i computed by $\mathcal{A}'$ satisfies

$$x_i = \mathcal{A}(f(\hat{x}), e^i) = \mathsf{gl}(\hat{x}, e^i) = \bigoplus_{j=1}^{n} \hat{x}_j \cdot e^i_j = \hat{x}_i.$$

Thus, $x_i = \hat{x}_i$ for all i and so $\mathcal{A}'$ outputs the correct inverse $x = \hat{x}$. ■

If f is one-way, it is impossible for any probabilistic polynomial-time algorithm to invert f with non-negligible probability. Thus, we conclude that there is no polynomial-time algorithm that always correctly computes $\mathsf{gl}(x,r)$ from $(f(x),r)$. This is a rather weak result that is very far from our ultimate goal of showing that $\mathsf{gl}(x,r)$ cannot be computed (with probability significantly better than 1/2) given $(f(x),r)$.

7.3.2 A More Involved Case

We now show that it is hard for any probabilistic polynomial-time algorithm $\mathcal{A}$ to compute $\mathsf{gl}(x,r)$ from $(f(x),r)$ with probability significantly better than 3/4. We will again show that any such $\mathcal{A}$ would imply the existence of a polynomial-time algorithm $\mathcal{A}'$ that inverts f with non-negligible probability. Notice that the strategy in the proof of Proposition 7.13 fails here because it may be that $\mathcal{A}$ *never* succeeds when $r = e^i$ (although it may succeed, say, on all other values of r). Furthermore, in the present case $\mathcal{A}'$ does not know if the result $\mathcal{A}(f(x),r)$ is equal to $\mathsf{gl}(x,r)$ or not; the only thing $\mathcal{A}'$ knows is that with high probability, algorithm $\mathcal{A}$ is correct. This further complicates the proof.

PROPOSITION 7.14 *Let f and gl be as in Theorem 7.12. If there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a polynomial $p(\cdot)$ such that*

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

for infinitely many values of n, then there exists a probabilistic polynomial-time algorithm $\mathcal{A}'$ such that

$$\Pr_{x\leftarrow\{0,1\}^n}\left[\mathcal{A}'(1^n,f(x))\in f^{-1}(f(x))\right]\geq\frac{1}{4\cdot p(n)}$$

for infinitely many values of n.

PROOF The main observation underlying the proof of this proposition is that for every $r\in\{0,1\}^n$, the values $\mathsf{gl}(x,r\oplus e^i)$ and $\mathsf{gl}(x,r)$ together can be used to compute the ith bit of x. (Recall that e^i denotes the n-bit string with 0s everywhere except the ith position.) This is true because

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

where $\bar{r}_i$ is the complement of r_i, and the second equality is due to the fact that for $j\neq i$, the value $x_j\cdot r_j$ appears in both sums and so is canceled out.

The above demonstrates that if $\mathcal{A}$ answers correctly on both $(f(x),r)$ and $(f(x),r\oplus e^i)$, then $\mathcal{A}'$ can correctly compute x_i. Unfortunately, $\mathcal{A}'$ does not know when $\mathcal{A}$ answers correctly and when it does not; $\mathcal{A}'$ knows only that $\mathcal{A}$ answers correctly with "high" probability. For this reason, $\mathcal{A}'$ will use multiple random values of r, using each one to obtain an estimate of x_i, and will then take the estimate occurring a majority of the time as its final guess for x_i.

As a preliminary step, we show that for many x's the probability that $\mathcal{A}$ answers correctly for both $(f(x), r)$ and $(f(x), r \oplus e^i)$, when r is uniform, is sufficiently high. This allows us to fix x and then focus solely on uniform choice of r, which makes the analysis easier.

CLAIM 7.15 *Let n be such that*

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

Then there exists a set $S_n\subseteq\{0,1\}^n$ of size at least $\frac{1}{2p(n)}\cdot 2^n$ such that for every $x\in S_n$ it holds that

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

PROOF Let $\varepsilon(n)=1/p(n)$, and define $S_n\subseteq\{0,1\}^n$ to be the set of all x's for which

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

We have

$$\begin{aligned}
\Pr_{x,r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big] &= \frac{1}{2^n}\sum_{x\in\{0,1\}^n}\Pr_{r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]\\
&= \frac{1}{2^n}\sum_{x\in S_n}\Pr_{r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]\\
&\quad+\frac{1}{2^n}\sum_{x\notin S_n}\Pr_{r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]\\
&\leq \frac{|S_n|}{2^n}+\frac{1}{2^n}\cdot\sum_{x\notin S_n}\left(\frac{3}{4}+\frac{\varepsilon(n)}{2}\right)\\
&\leq \frac{|S_n|}{2^n}+\left(\frac{3}{4}+\frac{\varepsilon(n)}{2}\right).
\end{aligned}$$

Since $\frac{3}{4}+\varepsilon(n)\leq\Pr_{x,r\leftarrow\{0,1\}^n}\Big[\mathcal{A}(f(x),r)=\mathsf{gl}(x,r)\Big]$, straightforward algebra gives $|S_n|\geq\frac{\varepsilon(n)}{2}\cdot 2^n$. ■

The following now follows as an easy consequence.

CLAIM 7.16 *Let n be such that*

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

Then there exists a set $S_n \subseteq \{0,1\}^n$ *of size at least* $\frac{1}{2p(n)} \cdot 2^n$ *such that for every* $x \in S_n$ *and every* i *it holds that*

$$\Pr_{r \leftarrow \{0,1\}^n}\left[\mathcal{A}(f(x),r) = \mathsf{gl}(x,r) \bigwedge \mathcal{A}(f(x), r \oplus e^i) = \mathsf{gl}(x, r \oplus e^i)\right] \geq \frac{1}{2} + \frac{1}{p(n)}\,.$$

PROOF Let $\varepsilon(n) = 1/p(n)$, and take S_n to be the set guaranteed by the previous claim. We know that for any $x \in S_n$ we have

$$\Pr_{r \leftarrow \{0,1\}^n}[\mathcal{A}(f(x),r) \neq \mathsf{gl}(x,r)] \leq \frac{1}{4} - \frac{\varepsilon(n)}{2}\,.$$

Fix $i \in \{1,\ldots,n\}$. If r is uniformly distributed then so is $r \oplus e^i$; thus

$$\Pr_{r \leftarrow \{0,1\}^n}[\mathcal{A}(f(x), r \oplus e^i) \neq \mathsf{gl}(x, r \oplus e^i)] \leq \frac{1}{4} - \frac{\varepsilon(n)}{2}\,.$$

We are interested in lower-bounding the probability that $\mathcal{A}$ outputs the correct answer for *both* $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x, r \oplus e^i)$; equivalently, we want to upper-bound the probability that $\mathcal{A}$ fails to output the correct answer in *either* of these cases. Note that r and $r \oplus e^i$ are not independent, so we cannot just multiply the probabilities of failure. However, we can apply the union bound (see Proposition A.7) and sum the probabilities of failure. That is, the probability that $\mathcal{A}$ is *incorrect* on either $\mathsf{gl}(x,r)$ or $\mathsf{gl}(x, r \oplus e^i)$ is at most

$$\left(\frac{1}{4} - \frac{\varepsilon(n)}{2}\right) + \left(\frac{1}{4} - \frac{\varepsilon(n)}{2}\right) = \frac{1}{2} - \varepsilon(n),$$

and so $\mathcal{A}$ is correct on *both* $\mathsf{gl}(x,r)$ and $\mathsf{gl}(x, r \oplus e^i)$ with probability *at least* $1/2 + \varepsilon(n)$. This proves the claim. ∎

For the rest of the proof we set $\varepsilon(n) = 1/p(n)$ and consider only those values of n for which

$$\Pr_{x,r \leftarrow \{0,1\}^n}\left[\mathcal{A}(f(x),r) = \mathsf{gl}(x,r)\right] \geq \frac{3}{4} + \varepsilon(n)\,. \tag{7.1}$$

The previous claim states that for an $\varepsilon(n)/2$ fraction of inputs x, and any i, algorithm $\mathcal{A}$ answers correctly on both $(f(x),r)$ and $(f(x), r \oplus e^i)$ with probability at least $1/2 + \varepsilon(n)$ over uniform choice of r, and from now on we focus only on such values of x. We construct a probabilistic polynomial-time algorithm $\mathcal{A}'$ that inverts $f(x)$ with probability at least $1/2$ when $x \in S_n$. This suffices to prove Proposition 7.14 since then, for infinitely many values of n,

$$\begin{aligned}
&\Pr_{x \leftarrow \{0,1\}^n}[\mathcal{A}'(1^n, f(x)) \in f^{-1}(f(x))] \\
&\quad \geq \Pr_{x \leftarrow \{0,1\}^n}[\mathcal{A}'(1^n, f(x)) \in f^{-1}(f(x)) \mid x \in S_n] \cdot \Pr_{x \leftarrow \{0,1\}^n}[x \in S_n] \\
&\quad \geq \frac{1}{2} \cdot \frac{\varepsilon(n)}{2} \;=\; \frac{1}{4p(n)}\,.
\end{aligned}$$

Algorithm $\mathcal{A}'$, given as input 1^n and y, works as follows:

1. For $i = 1, \ldots, n$ do:

   - Repeatedly choose a uniform $r \in \{0,1\}^n$ and compute $\mathcal{A}(y, r) \oplus \mathcal{A}(y, r \oplus e^i)$ as an "estimate" for the ith bit of the preimage of y. After doing this sufficiently many times (see below), let x_i be the "estimate" that occurs a majority of the time.

2. Output $x = x_1 \cdots x_n$.

We sketch an analysis of the probability that $\mathcal{A}'$ correctly inverts its given input y. (We allow ourselves to be a bit laconic, since a full proof for a more difficult case is given in the following section.) Say $y = f(\hat{x})$ and recall that we assume here that n is such that Equation (7.1) holds and $\hat{x} \in S_n$. Fix some i. The previous claim implies that the estimate $\mathcal{A}(y, r) \oplus \mathcal{A}(y, r \oplus e^i)$ is equal to $\mathsf{gl}(\hat{x}, e^i)$ with probability at least $\frac{1}{2} + \varepsilon(n)$ over choice of r. By obtaining sufficiently many estimates and letting x_i be the majority value, $\mathcal{A}'$ can ensure that x_i is equal to $\mathsf{gl}(\hat{x}, e^i)$ with probability at least $1 - \frac{1}{2n}$. Of course, we need to make sure that polynomially many estimates are enough. Fortunately, since $\varepsilon(n) = 1/p(n)$ for some polynomial p and an independent value of r is used for obtaining each estimate, the *Chernoff bound* (cf. Proposition A.14) shows that polynomially many estimates suffice.

Summarizing, we have that for each i the value x_i computed by $\mathcal{A}'$ is incorrect with probability at most $\frac{1}{2n}$. A union bound thus shows that $\mathcal{A}'$ is incorrect for *some* i with probability at most $n \cdot \frac{1}{2n} = \frac{1}{2}$. That is, $\mathcal{A}'$ is correct for all i—and thus correctly inverts y—with probability at least $1 - \frac{1}{2} = \frac{1}{2}$. This completes the proof of Proposition 7.14. ■

A corollary of Proposition 7.14 is that if f is a one-way function, then for any polynomial-time algorithm $\mathcal{A}$ the probability that $\mathcal{A}$ correctly guesses $\mathsf{gl}(x, r)$ when given $(f(x), r)$ is at most negligibly more than 3/4.

7.3.3 The Full Proof

We assume familiarity with the simplified proofs in the previous sections, and build on the ideas developed there. We rely on some terminology and standard results from probability theory discussed in Appendix A.3.

We prove the following proposition, which implies Theorem 7.12:

PROPOSITION 7.17 *Let f and gl be as in Theorem 7.12. If there exists a probabilistic polynomial-time algorithm $\mathcal{A}$ and a polynomial $p(\cdot)$ such that*

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

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

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

for infinitely many values of n.

PROOF Once again we set $\varepsilon(n) = 1/p(n)$ and consider only those values of n for which $\Pr_{x,r\leftarrow\{0,1\}^n}\left[\mathcal{A}(f(x), r) = \mathsf{gl}(x, r)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$. The following is analogous to Claim 7.15 and is proved in the same way.

CLAIM 7.18 *Let n be such that*

$$\Pr_{x,r\leftarrow\{0,1\}^n}\left[\mathcal{A}(f(x), r) = \mathsf{gl}(x, r)\right] \geq \frac{1}{2} + \varepsilon(n).$$

Then there exists a set $S_n \subseteq \{0,1\}^n$ of size at least $\frac{\varepsilon(n)}{2}\cdot 2^n$ such that for every $x \in S_n$ it holds that

$$\Pr_{r\leftarrow\{0,1\}^n}[\mathcal{A}(f(x), r) = \mathsf{gl}(x, r)] \geq \frac{1}{2} + \frac{\varepsilon(n)}{2}. \tag{7.2}$$

If we start by trying to prove an analogue of Claim 7.16, the best we can claim here is that when $x \in S_n$ we have

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

for any i. Thus, if we try to use $\mathcal{A}(f(x), r) \oplus \mathcal{A}(f(x), r \oplus e^i)$ as an estimate for x_i, all we can claim is that this estimate will be correct with probability at least $\varepsilon(n)$, which may not be any better than taking a random guess! We cannot claim that flipping the result gives a good estimate, either.

Instead, we design $\mathcal{A}'$ so that it computes $\mathsf{gl}(x, r)$ and $\mathsf{gl}(x, r\oplus e^i)$ by invoking $\mathcal{A}$ only once. We do this by having $\mathcal{A}'$ run $\mathcal{A}(x, r \oplus e^i)$, and having $\mathcal{A}'$ simply "guess" the value $\mathsf{gl}(x, r)$ itself. The naive way to do this would be to choose the r's independently, as before, and to have $\mathcal{A}'$ make an independent guess of $\mathsf{gl}(x, r)$ for each value of r. But then the probability that all such guesses are correct—which, as we will see, is necessary if $\mathcal{A}'$ is to output the correct inverse—would be negligible because polynomially many r's are used.

The crucial observation of the present proof is that $\mathcal{A}'$ can generate the r's in a *pairwise-independent* manner and make its guesses in a particular way so that with non-negligible probability all its guesses are correct. Specifically, in order to generate m values of r, we have $\mathcal{A}'$ select $\ell = \lceil \log(m + 1) \rceil$ independent and uniformly distributed strings $s^1, \ldots, s^\ell \in \{0,1\}^n$. Then, for every nonempty subset $I \subseteq \{1, \ldots, \ell\}$, we set $r^I := \oplus_{i\in I} s^i$. Since there are $2^\ell - 1$

nonempty subsets, this defines a collection of $2^{\lceil \log(m+1) \rceil} - 1 \geq m$ strings. Each such string is uniformly distributed. The strings are not independent, but they *are* pairwise independent. To see this, notice that for every two subsets $I \neq J$ there is an index $j \in I \cup J$ such that $j \notin I \cap J$. Without loss of generality, assume $j \notin I$. Then the value of s^j is uniform and independent of the value of r^I. Since s^j is included in the XOR that defines r^J, this implies that r^J is uniform and independent of r^I as well.

We now have the following two important observations:

1. Given $\mathsf{gl}(x, s^1), \ldots, \mathsf{gl}(x, s^\ell)$, it is possible to compute $\mathsf{gl}(x, r^I)$ for every subset $I \subseteq \{1, \ldots, \ell\}$. This is because

$$\mathsf{gl}(x, r^I) = \mathsf{gl}(x, \oplus_{i \in I} s^i) = \oplus_{i \in I} \mathsf{gl}(x, s^i).$$

2. If $\mathcal{A}'$ simply guesses the values of $\mathsf{gl}(x, s^1), \ldots, \mathsf{gl}(x, s^\ell)$ by choosing a uniform bit for each, then *all* these guesses will be correct with probability $1/2^\ell$. If m is polynomial in the security parameter n, then $1/2^\ell$ is not negligible, and so with *non-negligible probability* $\mathcal{A}'$ correctly guesses all the values $\mathsf{gl}(x, s^1), \ldots, \mathsf{gl}(x, s^\ell)$.

Combining the above yields a way of obtaining $m = \mathsf{poly}(n)$ uniform and pairwise-independent strings $\{r^I\}$ along with *correct* values for $\{\mathsf{gl}(x, r^I)\}$ with non-negligible probability. These values can then be used to compute x_i in the same way as in the proof of Proposition 7.14. Details follow.

The inversion algorithm $\mathcal{A}'$. We now provide a full description of an algorithm $\mathcal{A}'$ that receives inputs $1^n, y$ and tries to compute an inverse of y. The algorithm proceeds as follows:

1. Set $\ell := \lceil \log(2n/\varepsilon(n)^2 + 1) \rceil$.
2. Choose uniform, independent $s^1, \ldots, s^\ell \in \{0,1\}^n$ and $\sigma^1, \ldots, \sigma^\ell \in \{0,1\}$.
3. For every nonempty subset $I \subseteq \{1, \ldots, \ell\}$, compute $r^I := \oplus_{i \in I} s^i$ and $\sigma^I := \oplus_{i \in I} \sigma^i$.
4. For $i = 1, \ldots, n$ do:
   (a) For every nonempty subset $I \subseteq \{1, \ldots, \ell\}$, set
   $$x_i^I := \sigma^I \oplus \mathcal{A}(y, r^I \oplus e^i).$$
   (b) Set $x_i := \mathsf{majority}_I\{x_i^I\}$ (i.e., take the bit that appeared a majority of the time in the previous step).
5. Output $x = x_1 \cdots x_n$.

It remains to compute the probability that $\mathcal{A}'$ outputs $x \in f^{-1}(y)$. As in the proof of Proposition 7.14, we focus only on n as in Claim 7.18 and assume $y = f(\hat{x})$ for some $\hat{x} \in S_n$. Each σ^i represents a "guess" for the value of $\mathsf{gl}(\hat{x}, s^i)$. As noted earlier, with non-negligible probability all these guesses are correct; we show that conditioned on this event, $\mathcal{A}'$ outputs $x = \hat{x}$ with probability at least 1/2.

Assume $\sigma^i = \mathsf{gl}(\hat{x}, s^i)$ for all i. Then $\sigma^I = \mathsf{gl}(\hat{x}, r^I)$ for all I. Fix an index $i \in \{1, \ldots, n\}$ and consider the probability that $\mathcal{A}'$ obtains the correct value $x_i = \hat{x}_i$. For any nonempty I we have $\mathcal{A}(y, r^I \oplus e^i) = \mathsf{gl}(\hat{x}, r^I \oplus e^i)$ with probability at least $\frac{1}{2} + \varepsilon(n)/2$ over choice of r; this follows because $\hat{x} \in S_n$ and $r^I \oplus e^i$ is uniformly distributed. Thus, for any nonempty subset I we have $\Pr[x_i^I = \hat{x}_i] \geq \frac{1}{2} + \varepsilon(n)/2$. Moreover, the $\{x_i^I\}_{I \subseteq \{1,\ldots,\ell\}}$ are pairwise independent because the $\{r^I\}_{I \subseteq \{1,\ldots,\ell\}}$ (and hence the $\{r^I \oplus e^i\}_{I \subseteq \{1,\ldots,\ell\}}$) are pairwise independent. Since x_i is defined to be the value that occurs a majority of the time among the $\{x_i^I\}_{I \subseteq \{1,\ldots,\ell\}}$, we can apply Proposition A.13 to obtain

$$\begin{aligned}\Pr[x_i \neq \hat{x}_i] &\leq \frac{1}{4 \cdot (\varepsilon(n)/2)^2 \cdot (2^\ell - 1)} \\ &\leq \frac{1}{4 \cdot (\varepsilon(n)/2)^2 \cdot (2n/\varepsilon(n)^2)} \\ &= \frac{1}{2n}\,.\end{aligned}$$

The above holds for all i, so by applying a union bound we see that the probability that $x_i \neq \hat{x}_i$ for *some* i is at most 1/2. That is, $x_i = \hat{x}_i$ for *all* i (and hence $x = \hat{x}$) with probability at least 1/2.

Putting everything together: Let n be as in Claim 7.18 and $y = f(\hat{x})$. With probability at least $\varepsilon(n)/2$ we have $\hat{x} \in S_n$. All the guesses σ^i are correct with probability at least

$$\frac{1}{2^\ell} \geq \frac{1}{2 \cdot (2n/\varepsilon(n)^2 + 1)} > \frac{\varepsilon(n)^2}{5n}$$

for n sufficiently large. Conditioned on both the above, $\mathcal{A}'$ outputs $x = \hat{x}$ with probability at least 1/2. The overall probability with which $\mathcal{A}'$ inverts its input is thus at least $\varepsilon(n)^3/20n = 1/(20\, np(n)^3)$ for infinitely many n. Since $20\, np(n)^3$ is polynomial in n, this proves Proposition 7.17. ∎

7.4 Constructing Pseudorandom Generators

We first show how to construct pseudorandom generators that stretch their input by a single bit, under the assumption that one-way *permutations* exist. We then show how to extend this to obtain any polynomial expansion factor.

7.4.1 Pseudorandom Generators with Minimal Expansion

Let f be a one-way permutation with hard-core predicate hc. This means that $\mathsf{hc}(s)$ "looks random" given $f(s)$, when s is uniform. Furthermore, since f is a permutation, $f(s)$ itself is uniformly distributed. (Applying a permutation to a uniformly distributed value yields a uniformly distributed value.) So if s is a uniform n-bit string, the $(n+1)$-bit string $f(s)\|\mathsf{hc}(s)$ consists of a uniform n-bit string plus on additional bit that looks uniform even conditioned on the initial n bits; in other words, this $(n+1)$-bit string is *pseudorandom*. Thus, the algorithm G defined by $G(s) = f(s)\|\mathsf{hc}(s)$ is a pseudorandom generator.

THEOREM 7.19 *Let f be a one-way permutation with hard-core predicate hc. Then algorithm G defined by $G(s) = f(s)\|\mathsf{hc}(s)$ is a pseudorandom generator with expansion factor $\ell(n) = n+1$.*

PROOF Let D be a probabilistic polynomial-time algorithm. We prove that there is a negligible function negl such that

$$\Pr_{r\leftarrow\{0,1\}^{n+1}}[D(r)=1] - \Pr_{s\leftarrow\{0,1\}^{n}}[D(G(s))=1] \leq \mathsf{negl}(n). \tag{7.3}$$

A similar argument shows that there is a negligible function negl' for which

$$\Pr_{s\leftarrow\{0,1\}^{n}}[D(G(s))=1] - \Pr_{r\leftarrow\{0,1\}^{n+1}}[D(r)=1] \leq \mathsf{negl}'(n),$$

which completes the proof.

Observe first that

$$\begin{aligned}
\Pr_{r\leftarrow\{0,1\}^{n+1}}[D(r)=1] &= \Pr_{r\leftarrow\{0,1\}^{n},\, r'\leftarrow\{0,1\}}[D\left(r\|r'\right)=1] \\
&= \Pr_{s\leftarrow\{0,1\}^{n},\, r'\leftarrow\{0,1\}}[D\left(f(s)\|r'\right)=1] \\
&= \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^{n}}[D\left(f(s)\|\mathsf{hc}(s)\right)=1] \\
&\quad + \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^{n}}[D\left(f(s)\|\overline{\mathsf{hc}}(s)\right)=1],
\end{aligned}$$

using the fact that f is a permutation for the second equality, and that a uniform bit r' is equal to $\mathsf{hc}(s)$ with probability exactly $1/2$ for the third equality. Since

$$\Pr_{s\leftarrow\{0,1\}^{n}}[D(G(s))=1] = \Pr_{s\leftarrow\{0,1\}^{n}}[D\left(f(s)\|\mathsf{hc}(s)\right)=1]$$

(by definition of G), this means that Equation (7.3) is equivalent to

$$\frac{1}{2}\cdot\left(\Pr_{s\leftarrow\{0,1\}^{n}}[D\left(f(s)\|\overline{\mathsf{hc}}(s)\right)=1] - \Pr_{s\leftarrow\{0,1\}^{n}}[D\left(f(s)\|\mathsf{hc}(s)\right)=1]\right) \leq \mathsf{negl}(n).$$

Consider the following algorithm $\mathcal{A}$ that is given as input a value $y = f(s)$ and tries to predict the value of $\mathsf{hc}(s)$:

1. Choose uniform $r' \in \{0,1\}$.
2. Run $D(y\|r')$. If D outputs 0, output r'; otherwise output $\bar{r}'$.

Clearly $\mathcal{A}$ runs in polynomial time. By definition of $\mathcal{A}$, we have

$$\begin{aligned}
&\Pr_{s\leftarrow\{0,1\}^n}[\mathcal{A}(f(s)) = \mathsf{hc}(s)] \\
&\quad = \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^n}[\mathcal{A}(f(s)) = \mathsf{hc}(s) \mid r' = \mathsf{hc}(s)] \\
&\qquad + \frac{1}{2}\cdot \Pr_{s\leftarrow\{0,1\}^n}[\mathcal{A}(f(s)) = \mathsf{hc}(s) \mid r' \neq \mathsf{hc}(s)] \\
&\quad = \frac{1}{2}\cdot \left(\Pr_{s\leftarrow\{0,1\}^n}[D(f(s)\|\mathsf{hc}(s)) = 0] + \Pr_{s\leftarrow\{0,1\}^n}[D(f(s)\|\overline{\mathsf{hc}}(s)) = 1] \right) \\
&\quad = \frac{1}{2}\cdot \left(\left(1 - \Pr_{s\leftarrow\{0,1\}^n}[D(f(s)\|\mathsf{hc}(s)) = 1] \right) + \Pr_{s\leftarrow\{0,1\}^n}[D(f(s)\|\overline{\mathsf{hc}}(s)) = 1] \right) \\
&\quad = \frac{1}{2} + \frac{1}{2}\cdot \left(\Pr_{s\leftarrow\{0,1\}^n}[D(f(s)\|\overline{\mathsf{hc}}(s)) = 1] - \Pr_{s\leftarrow\{0,1\}^n}[D\left(f(s)\|\mathsf{hc}(s)\right) = 1] \right).
\end{aligned}$$

Since hc is a hard-core predicate of f, it follows that there exists a negligible function negl for which

$$\frac{1}{2}\cdot \left(\Pr_{s\leftarrow\{0,1\}^n}[D\left(f(s)\|\overline{\mathsf{hc}}(s)\right) = 1] - \Pr_{s\leftarrow\{0,1\}^n}[D\left(f(s)\|\mathsf{hc}(s)\right) = 1] \right) \leq \mathsf{negl}(n),$$

as desired. ■

7.4.2 Increasing the Expansion Factor

We now show that the expansion factor of a pseudorandom generator can be increased by any desired (polynomial) amount. This means that the previous construction, with expansion factor $\ell(n) = n + 1$, suffices for constructing a pseudorandom generator with arbitrary (polynomial) expansion factor.

THEOREM 7.20 *If there exists a pseudorandom generator G with expansion factor $n + 1$, then for any polynomial* poly *there exists a pseudorandom generator $\hat{G}$ with expansion factor* $\mathsf{poly}(n)$.

PROOF We first consider constructing a pseudorandom generator $\hat{G}$ that outputs $n + 2$ bits. $\hat{G}$ works as follows: Given an initial seed $s \in \{0,1\}^n$, it

computes $t_1 := G(s)$ to obtain $n+1$ pseudorandom bits. The initial n bits of t_1 are then used again as a seed for G; the resulting $n+1$ bits, concatenated with the final bit of t_1, yield the $(n+2)$-bit output. (See Figure 7.1.) The second application of G uses a pseudorandom seed rather than a random one. The proof of security we give next shows that this does not impact the pseudorandomness of the output.

We now prove that $\hat{G}$ is a pseudorandom generator. Define three sequences of distributions $\{H_n^0\}_{n=1,\ldots}$, $\{H_n^1\}_{n=1,\ldots}$, and $\{H_n^2\}_{n=1,\ldots}$, where each of H_n^0, H_n^1, and H_n^2 is a distribution on strings of length $n+2$. In distribution H_n^0, a uniform string $t_0 \in \{0,1\}^n$ is chosen and the output is $\hat{G}(t_0)$. In distribution H_n^1, a uniform string $t_1 \in \{0,1\}^{n+1}$ is chosen and parsed as $s_1 \| \sigma_1$ (where s_1 are the initial n bits of t_1 and σ_1 is the final bit). The output is $t_2 := G(s_1) \| \sigma_1$. In distribution H_n^2, the output is a uniform string $t_2 \in \{0,1\}^{n+2}$. We denote by $t_2 \leftarrow H_n^i$ the process of generating an $(n+2)$-bit string t_2 according to distribution H_n^i.

Fix an arbitrary probabilistic polynomial-time distinguisher D. We first claim that there is a negligible function negl' such that

$$\left| \Pr_{t_2 \leftarrow H_n^0}[D(t_2) = 1] - \Pr_{t_2 \leftarrow H_n^1}[D(t_2) = 1] \right| \leq \mathsf{negl}'(n). \tag{7.4}$$

To see this, consider the polynomial-time distinguisher D' that, on input $t_1 \in \{0,1\}^{n+1}$, parses t_1 as $s_1 \| \sigma_1$ with $|s_1| = n$, computes $t_2 := G(s_1) \| \sigma_1$, and outputs $D(t_2)$. Clearly D' runs in polynomial time. Observe that:

1. If t_1 is uniform, the distribution on t_2 generated by D' is exactly that of distribution H_n^1. Thus,

$$\Pr_{t_1 \leftarrow \{0,1\}^{n+1}}[D'(t_1) = 1] = \Pr_{t_2 \leftarrow H_n^1}[D(t_2) = 1].$$

2. If $t_1 = G(s)$ for uniform $s \in \{0,1\}^n$, the distribution on t_2 generated by D' is exactly that of distribution H_n^0. That is,

$$\Pr_{s \leftarrow \{0,1\}^n}[D'(G(s)) = 1] = \Pr_{t_2 \leftarrow H_n^0}[D(t_2) = 1].$$

Pseudorandomness of G implies that there is a negligible function negl' with

$$\left| \Pr_{s \leftarrow \{0,1\}^n}[D'(G(s)) = 1] - \Pr_{t_1 \leftarrow \{0,1\}^{n+1}}[D'(t_1) = 1] \right| \leq \mathsf{negl}'(n).$$

Equation (7.4) follows.

We next claim that there is a negligible function negl'' such that

$$\left| \Pr_{t_2 \leftarrow H_n^1}[D(t_2) = 1] - \Pr_{t_2 \leftarrow H_n^2}[D(t_2) = 1] \right| \leq \mathsf{negl}''(n). \tag{7.5}$$

To see this, consider the polynomial-time distinguisher D'' that, on input $w \in \{0,1\}^{n+1}$, chooses uniform $\sigma_1 \in \{0,1\}$, sets $t_2 := w\|\sigma_1$, and outputs $D(t_2)$. If w is uniform then so is t_2; thus,

$$\Pr_{w \leftarrow \{0,1\}^{n+1}}[D''(w) = 1] = \Pr_{t_2 \leftarrow H_n^2}[D(t_2) = 1].$$

On the other hand, if $w = G(s)$ for uniform $s \in \{0,1\}^n$, then t_2 is distributed exactly according to H_n^1 and so

$$\Pr_{s \leftarrow \{0,1\}^n}[D''(G(s)) = 1] = \Pr_{t_2 \leftarrow H_n^1}[D(t_2) = 1].$$

As before, pseudorandomness of G implies Equation (7.5).

Putting everything together, we have

$$\begin{aligned} &\left| \Pr_{s \leftarrow \{0,1\}^n}[D(\hat{G}(s)) = 1] - \Pr_{r \leftarrow \{0,1\}^{n+2}}[D(r) = 1] \right| \qquad (7.6) \\ &\quad = \left| \Pr_{t_2 \leftarrow H_n^0}[D(t_2) = 1] - \Pr_{t_2 \leftarrow H_n^2}[D(t_2) = 1] \right| \\ &\quad \leq \left| \Pr_{t_2 \leftarrow H_n^0}[D(t_2) = 1] - \Pr_{t_2 \leftarrow H_n^1}[D(t_2) = 1] \right| \\ &\qquad + \left| \Pr_{t_2 \leftarrow H_n^1}[D(t_2) = 1] - \Pr_{t_2 \leftarrow H_n^2}[D(t_2) = 1] \right| \\ &\quad \leq \mathsf{negl}'(n) + \mathsf{negl}''(n), \end{aligned}$$

using Equations (7.4) and (7.5). Since D was an arbitrary polynomial-time distinguisher, this proves that $\hat{G}$ is a pseudorandom generator.

The general case. The same idea as above can be iteratively applied to generate as many pseudorandom bits as desired. Formally, say we wish to construct a pseudorandom generator $\hat{G}$ with expansion factor $n + p(n)$, for some polynomial p. On input $s \in \{0,1\}^n$, algorithm $\hat{G}$ does (cf. Figure 7.1):

1. Set $t_0 := s$. For $i = 1, \ldots, p(n)$ do:
   (a) Let s_{i-1} be the first n bits of t_{i-1}, and let σ_{i-1} denote the remaining $i-1$ bits. (When $i = 1$, $s_0 = t_0$ and σ_0 is the empty string.)
   (b) Set $t_i := G(s'_{i-1})\|\sigma_{i-1}$.
2. Output $t_{p(n)}$.

We show that $\hat{G}$ is a pseudorandom generator. The proof uses a common technique known as a *hybrid argument*. (Actually, even the case of $p(n) = 2$, above, used a simple hybrid argument.) The main difference with respect to the previous proof is a technical one. Previously, we could define and explicitly work with three sequences of distributions $\{H_n^0\}$, $\{H_n^1\}$, and $\{H_n^2\}$. Here that is not possible since the number of distributions to consider grows with n.

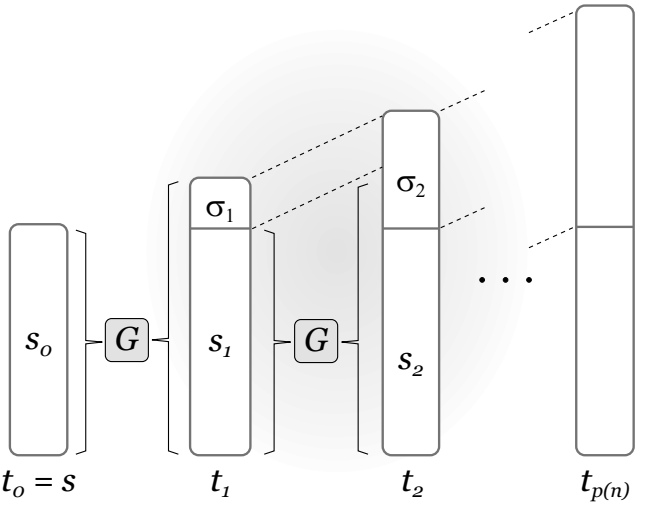


FIGURE 7.1: Increasing the expansion of a pseudorandom generator.

For any n and $0 \leq j \leq p(n)$, let H_n^j be the distribution on strings of length $n+p(n)$ defined as follows: choose uniform $t_j \in \{0,1\}^{n+j}$, then run $\hat{G}$ starting from iteration $j+1$ and output $t_{p(n)}$. (When $j = p(n)$ this means we simply choose uniform $t_{p(n)} \in \{0,1\}^{n+p(n)}$ and output it.) The crucial observation is that H_n^0 corresponds to outputting $\hat{G}(s)$ for uniform $s \in \{0,1\}^n$, while $H_n^{p(n)}$ corresponds to outputting a uniform $(n+p(n))$-bit string. Fixing any polynomial-time distinguisher D, this means that

$$\left| \Pr_{s \leftarrow \{0,1\}^n}[D(\hat{G}(s)) = 1] - \Pr_{r \leftarrow \{0,1\}^{n+p(n)}}[D(r) = 1] \right| = \left| \Pr_{t \leftarrow H_n^0}[D(t) = 1] - \Pr_{t \leftarrow H_n^{p(n)}}[D(t) = 1] \right|. \quad (7.7)$$

We prove the above is negligible, hence $\hat{G}$ is a pseudorandom generator.

Fix D as above, and consider the distinguisher D' that does the following when given a string $w \in \{0,1\}^{n+1}$ as input:

1. Choose uniform $j \in \{1, \ldots, p(n)\}$.
2. Choose uniform $\sigma'_j \in \{0,1\}^{j-1}$. (When $j = 1$ then σ'_j is the empty string.)
3. Set $t_j := w \| \sigma'_j$. Then run $\hat{G}$ starting from iteration $j+1$ to compute $t_{p(n)} \in \{0,1\}^{n+p(n)}$. Output $D(t_{p(n)})$.

Clearly D' runs in polynomial time. Analyzing the behavior of D' is more complicated than before, although the underlying ideas are the same. Fix n

and say D' chooses $j = j^*$. If w is uniform, then t_{j^*} is uniform and so the distribution on $t \stackrel{\text{def}}{=} t_{p(n)}$ is exactly that of distribution $H_n^{j^*}$. That is,

$$\Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1 \mid j = j^*] = \Pr_{t \leftarrow H_n^{j^*}}[D(t) = 1].$$

Since each value for j is chosen with equal probability,

$$\begin{aligned}\Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] &= \frac{1}{p(n)} \cdot \sum_{j^*=1}^{p(n)} \Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1 \mid j = j^*] \\ &= \frac{1}{p(n)} \cdot \sum_{j^*=1}^{p(n)} \Pr_{t \leftarrow H_n^{j^*}}[D(t) = 1]. \end{aligned} \tag{7.8}$$

On the other hand, say D' chooses $j = j^*$ and $w = G(s)$ for uniform $s \in \{0,1\}^n$. Defining $t_{j^*-1} = s \| \sigma'_{j^*}$, we see that t_{j^*-1} is uniform and so the experiment involving D' is equivalent to running $\hat{G}$ from iteration j^* to compute $t_{p(n)}$. That is, the distribution on $t \stackrel{\text{def}}{=} t_{p(n)}$ is now exactly that of distribution $H_n^{j^*-1}$, and so

$$\Pr_{s \leftarrow \{0,1\}^n}[D'(G(s)) = 1 \mid j = j^*] = \Pr_{t \leftarrow H_n^{j^*-1}}[D(t) = 1].$$

Therefore,

$$\begin{aligned}\Pr_{s \leftarrow \{0,1\}^n}[D'(\hat{G}(s)) = 1] &= \frac{1}{p(n)} \cdot \sum_{j^*=1}^{p(n)} \Pr_{s \leftarrow \{0,1\}^n}[D'(G(s)) = 1 \mid j = j^*] \\ &= \frac{1}{p(n)} \cdot \sum_{j^*=1}^{p(n)} \Pr_{t \leftarrow H_n^{j^*-1}}[D(t) = 1] \\ &= \frac{1}{p(n)} \cdot \sum_{j^*=0}^{p(n)-1} \Pr_{t \leftarrow H_n^{j^*}}[D(t) = 1]. \end{aligned} \tag{7.9}$$

We can now analyze how well D' distinguishes outputs of G from random:

$$\begin{aligned}&\left| \Pr_{s \leftarrow \{0,1\}^n}[D'(G(s)) = 1] - \Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] \right| \qquad (7.10) \\ &= \frac{1}{p(n)} \cdot \left| \sum_{j^*=0}^{p(n)-1} \Pr_{t \leftarrow H_n^{j^*}}[D(t) = 1] - \sum_{j^*=1}^{p(n)} \Pr_{t \leftarrow H_n^{j^*}}[D(t) = 1] \right| \\ &= \frac{1}{p(n)} \cdot \left| \Pr_{t \leftarrow H_n^0}[D(t) = 1] - \Pr_{t \leftarrow H_n^{p(n)}}[D(t) = 1] \right|, \end{aligned}$$

relying on Equations (7.8) and (7.9) for the first equality. (The second equality holds because the same terms are included in each sum, except for the first term of the left sum and the last term of the right sum.) Since G is a pseudorandom generator, the term on the left-hand side of Equation (7.10) is negligible; because p is polynomial, this implies that Equation (7.7) is negligible, completing the proof that $\hat{G}$ is a pseudorandom generator. ■

Putting it all together. Let f be a one-way permutation. Taking the pseudorandom generator with expansion factor $n+1$ from Theorem 7.19, and increasing the expansion factor to $n+\ell$ using the approach from the proof of Theorem 7.20, we obtain the following pseudorandom generator $\hat{G}$:

$$\hat{G}(s) = f^{(\ell)}(s) \,\|\, \mathsf{hc}(f^{(\ell-1)}(s)) \,\|\, \cdots \,\|\, \mathsf{hc}(s),$$

where $f^{(i)}(s)$ refers to i-fold iteration of f. Note that $\hat{G}$ uses ℓ evaluations of f, and generates one pseudorandom bit per evaluation using the hard-core predicate hc.

Connection to stream ciphers. Recall from Section 3.3.1 that a stream cipher (without an IV) is defined by algorithms $(\mathsf{Init}, \mathsf{GetBits})$, where Init takes a seed $s \in \{0,1\}^n$ and returns initial state st, and $\mathsf{GetBits}$ takes as input the current state st and outputs a bit σ and updated state st'. The construction $\hat{G}$ from the preceding proof fits nicely into this paradigm: take Init to be the trivial algorithm that outputs $\mathsf{st} = s$, and define $\mathsf{GetBits}(\mathsf{st})$ to compute $G(\mathsf{st})$, parse the result as $\mathsf{st}'\|\sigma$ with $|\mathsf{st}'| = n$, and output the bit σ and updated state st'. (If we use this stream cipher to generate $p(n)$ output bits starting from seed s, then we get exactly the final $p(n)$ bits of $\hat{G}(s)$ in reverse order.) The preceding proof shows that this yields a pseudorandom generator.

Hybrid arguments. A *hybrid argument* is a basic tool for proving indistinguishability when a basic primitive is (or several different primitives are) applied multiple times. Somewhat informally, the technique works by defining a series of intermediate "hybrid distributions" that bridge between two "extreme distributions" that we wish to prove indistinguishable. (In the proof above, these extreme distributions correspond to the output of $\hat{G}$ and a random string.) To apply the proof technique, three conditions should hold. First, the extreme distributions should match the original cases of interest. (In the proof above, H_n^0 was equal to the distribution induced by $\hat{G}$, while $H_n^{p(n)}$ was the uniform distribution.) Second, it must be possible to translate the capability of distinguishing consecutive hybrid distributions into breaking some underlying assumption. (Above, we essentially showed that distinguishing H_n^i from H_n^{i+1} was equivalent to distinguishing the output of G from random.) Finally, the number of hybrid distributions should be polynomial. See also Theorem 7.32.

7.5 Constructing Pseudorandom Functions

We now show how to construct a pseudorandom function from any (length-doubling) pseudorandom generator. Recall that a pseudorandom function is an efficiently computable, keyed function F that is indistinguishable from a truly random function in the sense described in Section 3.5.1. For simplicity, we restrict our attention here to the case where F is length preserving, meaning that for $k \in \{0,1\}^n$ the function F_k maps n-bit inputs to n-bit outputs. A (length-preserving) pseudorandom function can be viewed, informally, as a pseudorandom generator with expansion factor $n \cdot 2^n$; given such a pseudorandom generator G we could define $F_k(i)$ (for $0 \leq i < 2^n$) to be the ith n-bit block of $G(k)$. The reason this does not work is that F must be efficiently computable; there are exponentially many blocks, and we need a way to compute the ith block without having to compute all other blocks.

We will do this by computing "blocks" of the output by walking down a binary tree. We exemplify the construction by first showing a pseudorandom function taking 2-bit inputs. Let G be a pseudorandom generator with expansion factor $2n$. If we use G as in the proof of Theorem 7.20 we can obtain a pseudorandom generator $\hat{G}$ with expansion factor $4n$ that uses three invocations of G. (We produce n additional pseudorandom bits each time G is applied.) If we define $F'_k(i)$ (where $0 \leq i < 4$ and i is encoded as a 2-bit binary string) to be the ith block of $\hat{G}(k)$, then computation of $F'_k(3)$ would require computing all of $\hat{G}$ and hence three invocations of G. We show how to construct a pseudorandom function F using only two invocations of G on any input.

Let G_0 and G_1 be functions denoting the first and second halves of the output of G; i.e., $G(k) = G_0(k) \,\|\, G_1(k)$ where $|G_0(k)| = |G_1(k)| = |k|$. Define F as follows:

$$\begin{aligned} F_k(00) &= G_0(G_0(k)) \qquad & F_k(10) &= G_0(G_1(k)) \\ F_k(01) &= G_1(G_0(k)) \qquad & F_k(11) &= G_1(G_1(k)). \end{aligned}$$

We claim that the four strings above are pseudorandom *even when viewed together*. (This suffices to prove that F is pseudorandom.) Intuitively, this is because $G_0(k)\|G_1(k) = G(k)$ is pseudorandom and hence indistinguishable from a uniform $2n$-bit string $k_0\|k_1$. But then

$$G_0(G_0(k)) \,\|\, G_1(G_0(k)) \,\|\, G_0(G_1(k)) \,\|\, G_1(G_1(k))$$

is indistinguishable from

$$G_0(k_0) \,\|\, G_1(k_0) \,\|\, G_0(k_1) \,\|\, G_1(k_1) = G(k_0) \,\|\, G(k_1).$$

Since G is a pseudorandom generator, the above is indistinguishable from a uniform $4n$-bit string. A formal proof uses a hybrid argument.

Generalizing this idea, we can obtain a pseudorandom function on n-bit inputs by defining

$$F_k(x) = G_{x_n}(\cdots G_{x_1}(k)\cdots),$$

where $x = x_1 \cdots x_n$; see Construction 7.21. The intuition for why this function is pseudorandom is the same as before, but the formal proof is complicated by the fact that there are now exponentially many inputs to consider.

CONSTRUCTION 7.21

Let G be a pseudorandom generator with expansion factor $\ell(n) = 2n$, and define G_0, G_1 as in the text. For $k \in \{0,1\}^n$, define the function $F_k : \{0,1\}^n \to \{0,1\}^n$ as:

$$F_k(x_1 x_2 \cdots x_n) = G_{x_n}\left(\cdots(G_{x_2}(G_{x_1}(k)))\cdots\right).$$

A pseudorandom function from a pseudorandom generator.

It is useful to view this construction as defining, for each key $k \in \{0,1\}^n$, a complete binary tree of depth n in which each node contains an n-bit value. (See Figure 7.2, in which $n = 3$.) The root has value k, and for every internal node with value k' its left child has value $G_0(k')$ and its right child has value $G_1(k')$. The result $F_k(x)$ for $x = x_1 \cdots x_n$ is then defined to be the value on the leaf node reached by traversing the tree according to the bits of x, where $x_i = 0$ means "go left" and $x_i = 1$ means "go right." (The function is only defined for inputs of length n, and thus only values on the leaves are ever output.) The size of the tree is exponential in n. Nevertheless, to compute $F_k(x)$ the entire tree need not be constructed or stored; only n evaluations of G are needed.

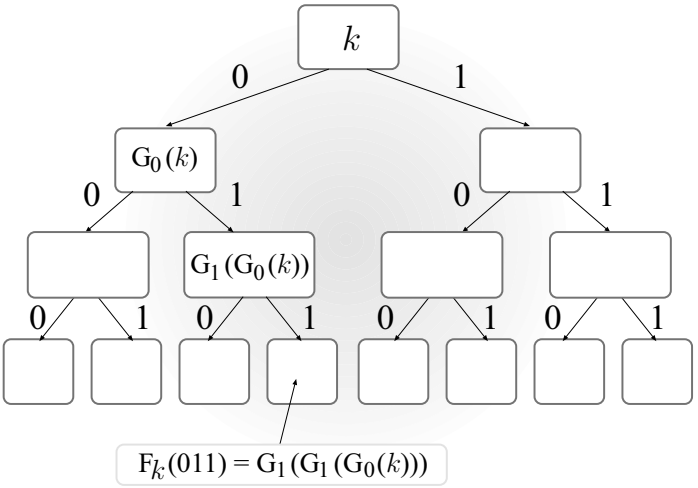


FIGURE 7.2: Constructing a pseudorandom function.

THEOREM 7.22 *If G is a pseudorandom generator with expansion factor $\ell(n) = 2n$, then Construction 7.21 is a pseudorandom function.*

PROOF We first show that for any polynomial t it is infeasible to distinguish $t(n)$ uniform $2n$-bit strings from $t(n)$ pseudorandom strings; i.e., for any polynomial t and any PPT algorithm A, the following is negligible:

$$\left| \Pr\left[A\left(r_1 \| \cdots \| r_{t(n)}\right) = 1\right] - \Pr\left[A\left(G(s_1) \| \cdots \| G(s_{t(n)})\right) = 1\right] \right|,$$

where the first probability is over uniform choice of $r_1, \ldots, r_{t(n)} \in \{0,1\}^{2n}$, and the second probability is over uniform choice of $s_1, \ldots, s_{t(n)} \in \{0,1\}^n$.

The proof is by a hybrid argument. Fix a polynomial t and a PPT algorithm A, and consider the following algorithm A':

Distinguisher A':
A' is given as input a string $w \in \{0,1\}^{2n}$.

1. Choose uniform $j \in \{1, \ldots, t(n)\}$.
2. Choose uniform, independent values $r_1, \ldots, r_{j-1} \in \{0,1\}^{2n}$ and $s_{j+1}, \ldots, s_{t(n)} \in \{0,1\}^n$.
3. Output $A\left(r_1 \| \cdots \| r_{j-1} \| w \| G(s_{j+1}) \| \cdots \| G(s_{t(n)})\right)$.

For any n and $0 \leq i \leq t(n)$, let G_n^i denote the distribution on strings of length $2n \cdot t(n)$ in which the first i "blocks" of length $2n$ are uniform and the remaining $t(n) - i$ blocks are pseudorandom. Note that $G_n^{t(n)}$ corresponds to the distribution in which all $t(n)$ blocks are uniform, while G_n^0 corresponds to the distribution in which all $t(n)$ blocks are pseudorandom. That is,

$$\begin{aligned} &\left| \Pr_{y \leftarrow G_n^{t(n)}}[A(y) = 1] - \Pr_{y \leftarrow G_n^0}[A(y) = 1] \right| \qquad (7.11) \\ &= \left| \Pr\left[A\left(r_1 \| \cdots \| r_{t(n)}\right) = 1\right] - \Pr\left[A\left(G(s_1) \| \cdots \| G(s_{t(n)})\right) = 1\right] \right| \end{aligned}$$

Say A' chooses $j = j^*$. If its input w is a uniform $2n$-bit string, then A is run on an input distributed according to $G_n^{j^*}$. If, on the other hand, $w = G(s)$ for uniform s, then A is run on an input distributed according to $G_n^{j^*-1}$. This means that

$$\Pr_{r \leftarrow \{0,1\}^{2n}}[A'(r) = 1] = \frac{1}{t(n)} \cdot \sum_{j=1}^{t(n)} \Pr_{y \leftarrow G_n^j}[A(y) = 1]$$

and

$$\Pr_{s \leftarrow \{0,1\}^n}[A'(G(s)) = 1] = \frac{1}{t(n)} \cdot \sum_{j=0}^{t(n)-1} \Pr_{y \leftarrow G_n^j}[A(y) = 1].$$

Therefore,

$$\left| \Pr_{r \leftarrow \{0,1\}^{2n}}[A'(r) = 1] - \Pr_{s \leftarrow \{0,1\}^{n}}[A'(G(s)) = 1] \right| \tag{7.12}$$
$$= \frac{1}{t(n)} \cdot \left| \Pr_{y \leftarrow G_n^{t(n)}}[A(y) = 1] - \Pr_{y \leftarrow G_n^0}[A(y) = 1] \right|.$$

Since G is a pseudorandom generator and A' runs in polynomial time, we know that the left-hand side of Equation (7.12) must be negligible; because $t(n)$ is polynomial, this implies that the left-hand side of Equation (7.11) is negligible as well.

Turning to the crux of the proof, we now show that F as in Construction 7.21 is a pseudorandom function. Let D be an arbitrary PPT distinguisher that is given 1^n as input. We show that D cannot distinguish between the case when it is given oracle access to a function that is equal to F_k for a uniform k, or a function chosen uniformly from Func_n. (See Section 3.5.1.) To do so, we use another hybrid argument. Here, we define a sequence of distributions over the values at the leaves of a complete binary tree of depth n. By associating each leaf with a string of length n as in Construction 7.21, we can equivalently view these as distributions over functions mapping n-bit inputs to n-bit outputs. For any n and $0 \leq i \leq n$, let H_n^i be the following distribution over the values at the leaves of a binary tree of depth n: first choose values for the nodes at level i independently and uniformly from $\{0,1\}^n$. Then for every node at level i or below with value k, its left child is given value $G_0(k)$ and its right child is given value $G_1(k)$. Note that H_n^n corresponds to the distribution in which all values at the leaves are chosen uniformly and independently, and thus corresponds to choosing a uniform function from Func_n, whereas H_n^0 corresponds to choosing a uniform key k in Construction 7.21 since in that case only the root (at level 0) is chosen uniformly. That is,

$$\left| \Pr_{k \leftarrow \{0,1\}^n}[D^{F_k(\cdot)}(1^n) = 1] - \Pr_{f \leftarrow \mathsf{Func}_n}[D^{f(\cdot)}(1^n) = 1] \right|$$
$$= \left| \Pr_{f \leftarrow H_n^0}[D^{f(\cdot)}(1^n) = 1] - \Pr_{f \leftarrow H_n^n}[D^{f(\cdot)}(1^n) = 1] \right|. \tag{7.13}$$

We show that Equation (7.13) is negligible, completing the proof.

Let $t = t(n)$ be a polynomial upper bound on the number of queries D makes to its oracle on input 1^n. Define a distinguisher A that tries to distinguish $t(n)$ uniform $2n$-bit strings from $t(n)$ pseudorandom strings, as follows:

Distinguisher A:
A is given as input a $2n \cdot t(n)$-bit string $w_1 \| \cdots \| w_{t(n)}$.

1. Choose uniform $j \in \{0, \ldots, n-1\}$. In what follows, A (implicitly) maintains a binary tree of depth n with n-bit values at (a subset of the) internal nodes at depth $j+1$ and below.

2. Run $D(1^n)$. When D makes oracle query $x = x_1 \cdots x_n$, look at the prefix $x_1 \cdots x_j$. There are two cases:
   - If D has never made a query with this prefix before, then use $x_1 \cdots x_j$ to reach a node v on the jth level of the tree. Take the next unused $2n$-bit string w and set the value of the left child of node v to the left half of w, and the value of the right child of v to the right half of w.
   - If D has made a query with prefix $x_1 \cdots x_j$ before, then node $x_1 \cdots x_{j+1}$ has already been assigned a value.

   Using the value at node $x_1 \cdots x_{j+1}$, compute the value at the leaf corresponding to $x_1 \cdots x_n$ as in Construction 7.21, and return this value to D.
3. When execution of D is done, output the bit returned by D.

A runs in polynomial time. It is important here that A does not need to store the entire binary tree of exponential size. Instead, it "fills in" the values of at most $2t(n)$ nodes in the tree. Say A chooses $j = j^*$. Observe that:

1. If A's input is a uniform $2n \cdot t(n)$-bit string, then the answers it gives to D are distributed exactly as if D were interacting with a function chosen from distribution $H_n^{j^*+1}$. This holds because the values of the nodes at level $j^* + 1$ of the tree are uniform and independent.

2. If A's input consists of $t(n)$ pseudorandom strings—i.e., $w_i = G(s_i)$ for uniform seed s_i—then the answers it gives to D are distributed exactly as if D were interacting with a function chosen from distribution $H_n^{j^*}$. This holds because the values of the nodes at level j^* of the tree (namely, the s-values) are uniform and independent. (These s-values are unknown to A, but this makes no difference.)

Proceeding as before, one can show that

$$\begin{aligned} &\left| \Pr\left[A\left(r_1 \| \cdots \| r_{t(n)}\right) = 1\right] - \Pr\left[A\left(G(s_1) \| \cdots \| G(s_{t(n)})\right) = 1\right] \right| \qquad (7.14) \\ &= \frac{1}{n} \cdot \left| \Pr_{f \leftarrow H_n^0}[D^{f(\cdot)}(1^n) = 1] - \Pr_{f \leftarrow H_n^n}[D^{f(\cdot)}(1^n) = 1] \right|. \end{aligned}$$

We have shown earlier that Equation (7.14) must be negligible. The above thus implies that Equation (7.13) must be negligible as well. ∎

7.6 Constructing (Strong) Pseudorandom Permutations

We next show how pseudorandom permutations and strong pseudorandom permutations can be constructed from any pseudorandom function. Recall

from Section 3.5.1 that a pseudorandom permutation is a pseudorandom function that is also efficiently invertible, while a *strong* pseudorandom permutation is additionally hard to distinguish from a random permutation even by an adversary given oracle access to both the permutation *and its inverse*.

Feistel networks revisited. A Feistel network, introduced in Section 6.2.2, provides a way of constructing an invertible function from an arbitrary set of functions. A Feistel network operates in a series of rounds. The input to the ith round is a string of length $2n$, divided into two n-bit halves L_{i-1} and R_{i-1} (the "left half" and the "right half," respectively). The output of the ith round is the $2n$-bit string (L_i, R_i) where

$$L_i := R_{i-1} \quad \text{and} \quad R_i := L_{i-1} \oplus f_i(R_{i-1})$$

for some efficiently computable (but not necessarily invertible) function f_i mapping n-bit inputs to n-bit outputs. We denote by $\mathsf{Feistel}_{f_1,\ldots,f_r}$ the r-round Feistel network using functions $f_1, \ldots, f_r$. (That is, $\mathsf{Feistel}_{f_1,\ldots,f_r}(L_0, R_0)$ outputs the $2n$-bit string (L_r, R_r).) We saw in Section 6.2.2 that $\mathsf{Feistel}_{f_1,\ldots,f_r}$ is an efficiently invertible permutation regardless of the $\{f_i\}$.

We can define a keyed permutation by using a Feistel network in which the $\{f_i\}$ depend on a key. For example, let $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a pseudorandom function, and define the keyed *permutation* $F^{(1)}$ as

$$F^{(1)}_k(x) \stackrel{\text{def}}{=} \mathsf{Feistel}_{F_k}(x).$$

(Note that $F^{(1)}_k$ has an n-bit key and maps $2n$-bit inputs to $2n$-bit outputs.) Is $F^{(1)}$ pseudorandom? A little thought shows that it is decidedly *not*. For any key $k \in \{0,1\}^n$, the first n bits of the output of $F^{(1)}_k$ (that is, L_1) are equal to the last n bits of the input (i.e., R_0), something that occurs with only negligible probability for a random function.

Trying again, define $F^{(2)} : \{0,1\}^{2n} \times \{0,1\}^{2n} \to \{0,1\}^{2n}$ as follows:

$$F^{(2)}_{k_1,k_2}(x) \stackrel{\text{def}}{=} \mathsf{Feistel}_{F_{k_1},F_{k_2}}(x). \tag{7.15}$$

(Note that k_1 and k_2 are independent keys.) Unfortunately, $F^{(2)}$ is not pseudorandom either, as you are asked to show in Exercise 7.16.

Given this, it may be somewhat surprising that a *three*-round Feistel network *is* pseudorandom. Define the keyed permutation $F^{(3)}$, taking a key of length $3n$ and mapping $2n$-bit inputs to $2n$-bit outputs, as follows:

$$F^{(3)}_{k_1,k_2,k_3}(x) \stackrel{\text{def}}{=} \mathsf{Feistel}_{F_{k_1},F_{k_2},F_{k_3}}(x) \tag{7.16}$$

where, once again, k_1, k_2, and k_3 are independent. We have:

THEOREM 7.23 *If F is a pseudorandom function, then $F^{(3)}$ is a pseudorandom permutation.*

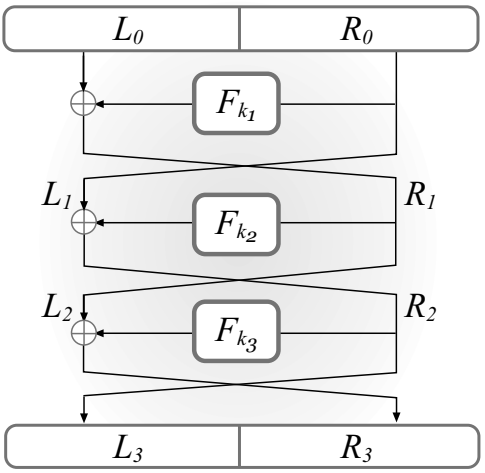


FIGURE 7.3: A three-round Feistel network, as used to construct a pseudorandom permutation from a pseudorandom function.

PROOF In the standard way, we can replace the pseudorandom functions used in the construction of $F^{(3)}$ with functions chosen uniformly at random instead. Pseudorandomness of F implies that this has only a negligible effect on the output of any probabilistic polynomial-time distinguisher interacting with $F^{(3)}$ as an oracle. We leave the details as an exercise.

Let D be a probabilistic polynomial-time distinguisher. In the remainder of the proof, we show the following is negligible:

$$\left|\Pr[D^{\mathsf{Feistel}_{f_1,f_2,f_3}(\cdot)}(1^n) = 1] - \Pr[D^{\pi(\cdot)}(1^n) = 1]\right|,$$

where the first probability is taken over uniform and independent choice of f_1, f_2, f_3 from Func_n, and the second probability is taken over uniform choice of π from Perm_{2n}. Fix some value for the security parameter n, and let $q = q(n)$ denote a polynomial upper bound on the number of oracle queries made by D. We assume without loss of generality that D never makes the same oracle query twice. Focusing on D's interaction with $\mathsf{Feistel}_{f_1,f_2,f_3}(\cdot)$, let (L_0^i, R_0^i) denote the ith query D makes to its oracle, and let (L_1^i, R_1^i), (L_2^i, R_2^i), and (L_3^i, R_3^i) denote the intermediate values after rounds 1, 2, and 3, respectively, that result from that query. (See Figure 7.3.) Note that D chooses (L_0^i, R_0^i) and sees the result (L_3^i, R_3^i), but does not directly observe (L_1^i, R_1^i) or (L_2^i, R_2^i).

We say there is *a collision at* R_1 if $R_1^i = R_1^j$ for some distinct i, j. We first prove that a collision at R_1 occurs with only negligible probability. Consider any fixed, distinct i, j. If $R_0^i = R_0^j$ then $L_0^i \neq L_0^j$, but then

$$R_1^i = L_0^i \oplus f_1(R_0^i) \neq L_0^j \oplus f_1(R_0^j) = R_1^j.$$

If $R_0^i \neq R_0^j$ then $f_1(R_0^i)$ and $f_1(R_0^j)$ are uniform and independent, so

$$\Pr\left[L_0^i \oplus f_1(R_0^i) = L_0^j \oplus f_1(R_0^j)\right] = \Pr\left[f_1(R_0^j) = L_0^i \oplus f_1(R_0^i) \oplus L_0^j\right] = 2^{-n}.$$

Taking a union bound over all distinct i, j shows that the probability of a collision at R_1 is at most $q^2/2^n$.

Say there is *a collision at* R_2 if $R_2^i = R_2^j$ for some distinct i, j. We prove that conditioned on no collision at R_1, the probability of a collision at R_2 is negligible. The analysis is as above: consider any fixed i, j, and note that if there is no collision at R_1 then $R_1^i \neq R_1^j$. Thus $f_2(R_1^i)$ and $f_2(R_1^j)$ are uniform and independent, and therefore

$$\Pr\left[L_1^i \oplus f_2(R_1^i) = L_1^j \oplus f_2(R_1^j) \mid \text{no collision at } R_1\right] = 2^{-n}.$$

(Note that f_2 is independent of f_1, making the above calculation easy.) Taking a union bound over all distinct i, j gives

$$\Pr[\text{collision at } R_2 \mid \text{no collision at } R_1] \leq q^2/2^n.$$

Note that $L_3^i = R_2^i = L_1^i \oplus f_2(R_1^i)$; so, conditioned on there being no collision at R_1, the values $L_3^1, \ldots, L_3^q$ are all independent and uniformly distributed in $\{0,1\}^n$. If we additionally condition on the event that there is no collision at R_2, then the values $L_3^1, \ldots, L_3^q$ are uniformly distributed among all sequences of q *distinct* values in $\{0,1\}^n$. Similarly, $R_3^i = L_2^i \oplus f_3(R_2^i)$; thus, conditioned on there being no collision at R_2, the values $R_3^1, \ldots, R_3^q$ are all uniformly distributed in $\{0,1\}^n$, independent of each other as well as $L_3^1, \ldots, L_3^q$.

To summarize: when querying $F^{(3)}$ (with uniform round functions) on a series of q distinct inputs, except with negligible probability the output values (L_3^1, R_3^1), $\ldots$, (L_3^q, R_3^q) are distributed such that the $\{L_3^i\}$ are uniform and independent, but distinct, n-bit values, and the $\{R_3^i\}$ are uniform and independent n-bit values. In contrast, when querying a random permutation on a series of q distinct inputs, the output values (L_3^1, R_3^1), $\ldots$, (L_3^q, R_3^q) are uniform and independent, but distinct, $2n$-bit values. The best distinguishing attack for D, then, is to guess that it is interacting with a random permutation if $L_3^i = L_3^j$ for some distinct i, j. But that event occurs with negligible probability even in that case. This can be turned into a formal proof. ■

$F^{(3)}$ is not a *strong* pseudorandom permutation, as you are asked to demonstrate in Exercise 7.17. Fortunately, adding a fourth round *does* yield a strong pseudorandom permutation. The details are given as Construction 7.24.

THEOREM 7.25 *If F is a pseudorandom function, then Construction 7.24 is a strong pseudorandom permutation that maps $2n$-bit inputs to $2n$-bit outputs (and uses a $4n$-bit key).*

CONSTRUCTION 7.24

Let F be a keyed, length-preserving function. Define the keyed permutation $F^{(4)}$ as follows:

- **Inputs:** A key $k = (k_1, k_2, k_3, k_4)$ with $|k_i| = n$, and an input $x \in \{0,1\}^{2n}$ parsed as (L_0, R_0) with $|L_0| = |R_0| = n$.
- **Computation:**
  1. Compute $L_1 := R_0$ and $R_1 := L_0 \oplus F_{k_1}(R_0)$.
  2. Compute $L_2 := R_1$ and $R_2 := L_1 \oplus F_{k_2}(R_1)$.
  3. Compute $L_3 := R_2$ and $R_3 := L_2 \oplus F_{k_3}(R_2)$.
  4. Compute $L_4 := R_3$ and $R_4 := L_3 \oplus F_{k_4}(R_3)$.
  5. Output (L_4, R_4).

A strong pseudorandom permutation from any pseudorandom function.

7.7 Assumptions for Private-Key Cryptography

We have shown that (1) if there exist one-way permutations, then there exist pseudorandom generators; (2) if there exist pseudorandom generators, then there exist pseudorandom functions; and (3) if there exist pseudorandom functions, then there exist (strong) pseudorandom permutations. Although we did not prove it here, it is possible to construct pseudorandom generators from one-way *functions*. We thus have the following fundamental theorem:

THEOREM 7.26 *If one-way functions exist, then so do pseudorandom generators, pseudorandom functions, and strong pseudorandom permutations.*

All the private-key schemes we have studied in Chapters 3 and 4 can be constructed from pseudorandom generators/functions. We therefore have:

THEOREM 7.27 *If one-way functions exist, then so do CCA-secure private-key encryption schemes and secure message authentication codes.*

That is, *one-way functions are sufficient for all private-key cryptography.* Here, we show that one-way functions are also *necessary.*

Pseudorandomness implies one-way functions. We begin by showing that pseudorandom generators imply the existence of one-way functions:

PROPOSITION 7.28 *If a pseudorandom generator exists, then so does a one-way function.*

PROOF Let G be a pseudorandom generator with expansion factor $\ell(n) = 2n$. (By Theorem 7.20, we know that the existence of a pseudorandom generator implies the existence of one with this expansion factor.) We show that G itself is one-way. Efficient computability is straightforward (since G can be computed in polynomial time). We show that the ability to invert G can be translated into the ability to distinguish the output of G from uniform. Intuitively, this holds because the ability to invert G implies the ability to find the seed used by the generator.

Let $\mathcal{A}$ be an arbitrary probabilistic polynomial-time algorithm. We show that $\Pr[\mathsf{Invert}_{\mathcal{A},G}(n) = 1]$ is negligible (cf. Definition 7.1). To see this, consider the following PPT distinguisher D: on input a string $w \in \{0,1\}^{2n}$, run $\mathcal{A}(w)$ to obtain output s. If $G(s) = w$ then output 1; otherwise, output 0.

We now analyze the behavior of D. First consider the probability that D outputs 1 when its input string w is uniform. Since there are at most 2^n values in the range of G (namely, the values $\{G(s)\}_{s\in\{0,1\}^n}$), the probability that w is in the range of G is at most $2^n/2^{2n} = 2^{-n}$. When w is not in the range of G, it is impossible for $\mathcal{A}$ to compute an inverse of w and thus impossible for D to output 1. We conclude that

$$\Pr_{w\leftarrow\{0,1\}^{2n}}[D(w) = 1] \leq 2^{-n}.$$

On the other hand, if $w = G(s)$ for a seed $s \in \{0,1\}^n$ chosen uniformly at random then, by definition, $\mathcal{A}$ computes a correct inverse (and so D outputs 1) with probability exactly equal to $\Pr[\mathsf{Invert}_{\mathcal{A},G}(n) = 1]$. Thus,

$$\left|\Pr_{w\leftarrow\{0,1\}^{2n}}[D(w) = 1] - \Pr_{s\leftarrow\{0,1\}^{n}}[D(G(s)) = 1]\right| \geq \Pr[\mathsf{Invert}_{\mathcal{A},G}(n) = 1] - 2^{-n}.$$

Since G is a pseudorandom generator, the above must be negligible. Since 2^{-n} is negligible, this implies that $\Pr[\mathsf{Invert}_{\mathcal{A},G}(n) = 1]$ is negligible as well and so G is one-way. ■

Non-trivial private-key encryption implies one-way functions. Proposition 7.28 does not imply that one-way functions are needed for constructing secure private-key encryption schemes, since it may be possible to construct the latter without relying on a pseudorandom generator. Furthermore, it *is* possible to construct perfectly secret encryption schemes (see Chapter 2), as long as the plaintext is no longer than the key. Thus, a proof that secure private-key encryption implies one-way functions requires more care.

PROPOSITION 7.29 *If there exists an EAV-secure private-key encryption scheme that encrypts messages twice as long as its key, then a one-way function exists.*

PROOF Let $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme that has indistinguishable encryptions in the presence of an eavesdropper and encrypts messages of length $2n$ when the key has length n. (We assume for simplicity that the key is chosen uniformly.) Say that when an n-bit key is used, Enc uses at most $\ell(n)$ bits of randomness. Denote the encryption of a message m using key k and randomness r by $\mathsf{Enc}_k(m;r)$.

Define the following function f:

$$f(k,m,r) \stackrel{\text{def}}{=} \mathsf{Enc}_k(m;r) \,\|\, m,$$

where $|k| = n$, $|m| = 2n$, and $|r| = \ell(n)$. We claim that f is a one-way function. Clearly it can be efficiently computed; we show that it is hard to invert. Letting $\mathcal{A}$ be an arbitrary PPT algorithm, we show that $\Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1]$ is negligible (cf. Definition 7.1).

Consider the following probabilistic polynomial-time adversary $\mathcal{A}'$ attacking private-key encryption scheme Π (i.e., in experiment $\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n)$):

Adversary $\mathcal{A}'(1^n)$

1. Choose uniform $m_0, m_1 \leftarrow \{0,1\}^{2n}$ and output them. Receive in return a challenge ciphertext c.
2. Run $\mathcal{A}(c \,\|\, m_0)$ to obtain (k', m', r'). If $f(k', m', r') = c \,\|\, m_0$, output 0; else, output 1.

We now analyze the behavior of $\mathcal{A}'$. When c is an encryption of m_0, then $c \| m_0$ is distributed exactly as $f(k, m_0, r)$ for uniform k, m_0, and r. Therefore, $\mathcal{A}$ outputs a valid inverse of $c \| m_0$ (and hence $\mathcal{A}'$ outputs 0) with probability exactly equal to $\Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1]$.

On the other hand, when c is an encryption of m_1 then c is independent of m_0. For any fixed value of the challenge ciphertext c, there are at most 2^n possible messages (one for each possible key) to which c can correspond. Since m_0 is a uniform $2n$-bit string, this means the probability there *exists* some key k for which $\mathsf{Dec}_k(c) = m_0$ is at most $2^n/2^{2n} = 2^{-n}$. This gives an upper bound on the probability with which $\mathcal{A}$ can possibly output a valid inverse of $c \,\|\, m_0$ under f, and hence an upper bound on the probability with which $\mathcal{A}'$ outputs 0 in that case.

Putting the above together, we have:

$$\begin{aligned}
&\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1\right] \\
&= \frac{1}{2} \cdot \Pr\left[\mathcal{A}' \text{ outputs } 0 \mid b = 0\right] + \frac{1}{2} \cdot \Pr\left[\mathcal{A}' \text{ outputs } 1 \mid b = 1\right] \\
&\geq \frac{1}{2} \cdot \Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1] + \frac{1}{2} \cdot \left(1 - 2^{-n}\right) \\
&= \frac{1}{2} + \frac{1}{2} \cdot \left(\Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1] - 2^{-n}\right) .
\end{aligned}$$

Security of Π means that $\Pr\left[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(n)$ for some negligible function negl. This, in turn, implies that $\Pr[\mathsf{Invert}_{\mathcal{A},f}(n) = 1]$ is negligible, completing the proof that f is one-way. ■

Message authentication codes imply one-way functions. It is also true that message authentication codes satisfying Definition 4.2 imply the existence of one-way functions. As in the case of private-key encryption, a proof of this fact is somewhat subtle because unconditional message authentication codes *do* exist when there is an *a priori* bound on the number of messages that will be authenticated. (See Section 4.6.) Thus, a proof relies on the fact that Definition 4.2 requires security even when the adversary sees the authentication tags of an *arbitrary* (polynomial) number of messages. The proof is somewhat involved, so we do not give it here.

Discussion. We conclude that the existence of one-way functions is necessary and sufficient for all (non-trivial) private-key cryptography. In other words, one-way functions are a *minimal* assumption as far as private-key cryptography is concerned. Interestingly, this appears not to be the case for hash functions and public-key encryption, where one-way functions are known to be necessary but are not known (or believed) to be sufficient.

7.8 Computational Indistinguishability

The notion of computational indistinguishability is central to the theory of cryptography, and it underlies much of what we have seen in Chapter 3 and this chapter. Informally, two probability distributions are *computationally indistinguishable* if no efficient algorithm can tell them apart (or *distinguish* them). In more detail, consider two distributions X and Y over strings of some length ℓ; that is, X and Y each assigns some probability to every string in $\{0,1\}^\ell$. When we say that some algorithm D cannot distinguish these two distributions, we mean that D cannot tell whether it is given a string sampled according to distribution X or whether it is given a string sampled according to distribution Y. Put differently, if we imagine D outputting "0" when it believes its input was sampled according to X and outputting "1" if it thinks its input was sampled according to Y, then the probability that D outputs "1" should be roughly the same regardless of whether D is provided with a sample from X or from Y. In other words, we want

$$\left| \Pr_{s \leftarrow X}[D(s) = 1] - \Pr_{s \leftarrow Y}[D(s) = 1] \right|$$

to be small.

This should be reminiscent of the way we defined pseudorandom generators and, indeed, we will soon formally redefine the notion of a pseudorandom generator using this terminology.

The formal definition of computational indistinguishability refers to *probability ensembles*, which are infinite sequences of probability distributions. (This formalism is necessary for a meaningful asymptotic approach.) Although the notion can be generalized, for our purposes we consider probability ensembles in which the underlying distributions are indexed by natural numbers. If for every natural number n we have a distribution X_n, then $\mathcal{X} = \{X_n\}_{n\in\mathbb{N}}$ is a probability ensemble. It is often the case that $X_n = Y_{t(n)}$ for some function t, in which case we write $\{Y_{t(n)}\}_{n\in\mathbb{N}}$ in place of $\{X_n\}_{n\in\mathbb{N}}$.

We will only be interested in *efficiently sampleable* probability ensembles. An ensemble $\mathcal{X} = \{X_n\}_{n\in\mathbb{N}}$ is efficiently sampleable if there is a probabilistic polynomial-time algorithm S such that the random variables $S(1^n)$ and X_n are identically distributed. That is, algorithm S is an efficient way of sampling $\mathcal{X}$.

We can now formally define what it means for two ensembles to be computationally indistinguishable.

DEFINITION 7.30 *Two probability ensembles $\mathcal{X} = \{X_n\}_{n\in\mathbb{N}}$ and $\mathcal{Y} = \{Y_n\}_{n\in\mathbb{N}}$ are* computationally indistinguishable, *denoted $\mathcal{X} \stackrel{c}{\equiv} \mathcal{Y}$, if for every probabilistic polynomial-time distinguisher D there exists a negligible function* negl *such that:*

$$\left| \Pr_{x\leftarrow X_n}[D(1^n, x) = 1] - \Pr_{y\leftarrow Y_n}[D(1^n, y) = 1] \right| \leq \mathsf{negl}(n).$$

In the definition, D is given the unary input 1^n so it can run in time polynomial in n. This is important when the outputs of X_n and Y_n may have length less than n. As shorthand in probability expressions, we will sometimes write X as a placeholder for a random sample from distribution X. That is, we would write $\Pr[D(1^n, X_n) = 1]$ in place of $\Pr_{x\leftarrow X_n}[D(1^n, x) = 1]$.

The relation of computational indistinguishability is transitive: if $\mathcal{X} \stackrel{c}{\equiv} \mathcal{Y}$ and $\mathcal{Y} \stackrel{c}{\equiv} \mathcal{Z}$, then $\mathcal{X} \stackrel{c}{\equiv} \mathcal{Z}$.

Pseudorandomness and pseudorandom generators. Pseudorandomness is just a special case of computational indistinguishability. For any integer ℓ, let U_ℓ denote the uniform distribution over $\{0,1\}^\ell$. We can define a pseudorandom generator as follows:

DEFINITION 7.31 *Let $\ell(\cdot)$ be a polynomial and let G be a (deterministic) polynomial-time algorithm where for all s it holds that $|G(s)| = \ell(|s|)$. We say that G is a* pseudorandom generator *if the following two conditions hold:*

1. **(Expansion:)** *For every n it holds that $\ell(n) > n$.*

2. **(Pseudorandomness:)** *The ensemble* $\{G(U_n)\}_{n\in\mathbb{N}}$ *is computationally indistinguishable from the ensemble* $\{U_{\ell(n)}\}_{n\in\mathbb{N}}$.

Many of the other definitions and assumptions in this book can also be cast as special cases or variants of computational indistinguishability.

Multiple samples. An important theorem regarding computational indistinguishability is that polynomially many samples of (efficiently sampleable) computationally indistinguishable ensembles are also computationally indistinguishable.

THEOREM 7.32 *Let* $\mathcal{X}$ *and* $\mathcal{Y}$ *be efficiently sampleable probability ensembles that are computationally indistinguishable. Then, for every polynomial* p*, the ensemble* $\overline{\mathcal{X}} = \{(X_n^{(1)}, \ldots, X_n^{(p(n))})\}_{n\in\mathbb{N}}$ *is computationally indistinguishable from the ensemble* $\overline{\mathcal{Y}} = \{(Y_n^{(1)}, \ldots, Y_n^{(p(n))})\}_{n\in\mathbb{N}}$.

For example, let G be a pseudorandom generator with expansion factor $2n$, in which case the ensembles $\{G(U_n)\}_{n\in\mathbb{N}}$ and $\{U_{2n}\}_{n\in\mathbb{N}}$ are computationally indistinguishable. In the proof of Theorem 7.22 we showed that for any polynomial t the ensembles

$$\{(\underbrace{G(U_n), \ldots, G(U_n)}_{t(n)})\}_{n\in\mathbb{N}} \quad \text{and} \quad \{(\underbrace{U_{2n}, \ldots, U_{2n}}_{t(n)})\}_{n\in\mathbb{N}}$$

are also computationally indistinguishable. Theorem 7.32 is proved by a hybrid argument in exactly the same way.

References and Additional Reading

The notion of a one-way function was first proposed by Diffie and Hellman [58] and later formalized by Yao [179]. Hard-core predicates were introduced by Blum and Micali [37], and the fact that there exists a hard-core predicate for every one-way function was proved by Goldreich and Levin [79].

The first construction of pseudorandom generators (under a specific number-theoretic hardness assumption) was given by Blum and Micali [37]. The construction of a pseudorandom generator from any one-way permutation was given by Yao [179], and the result that pseudorandom generators can be constructed from any one-way function was shown by Håstad et al. [85]. Pseudorandom functions were defined and constructed by Goldreich, Goldwasser and Micali [78] and their extension to (strong) pseudorandom permutations was shown by Luby and Rackoff [116]. The fact that one-way functions are a