

Third Edition

Introduction to **CRYPTOGRAPHY** with Coding Theory



Wade Trappe
Lawrence C. Washington

Introduction to Cryptography

with Coding Theory

3rd edition

Wade Trappe

Wireless Information Network Laboratory and the
Electrical and Computer Engineering Department
Rutgers University

Lawrence C. Washington

Department of Mathematics University of Maryland

Portfolio Manager: *Chelsea Kharakozoua*

Content Manager: *Jeff Weidenaar*

Content Associate: *Jonathan Krebs*

Content Producer: *Tara Corpuz*

Managing Producer: *Scott Disanno*

Producer: *Jean Choe*

Manager, Courseware QA: *Mary Durnwald*

Product Marketing Manager: *Stacey Sveum*

Product and Solution Specialist: *Rosemary Morten*

Senior Author Support/Technology Specialist:
Joe Vetere

Manager, Rights and Permissions: *Gina Cheskka*

**Text and Cover Design, Production
Coordination, Composition, and Illustrations:**
Integra Software Services Pvt. Ltd

Manufacturing Buyer: *Carol Melville, LSC
Communications*

Cover Image: *Photographer is my life/Getty Images*

**Copyright © 2020, 2006, 2002 by Pearson
Education, Inc. 221 River Street, Hoboken, NJ
07030.** All Rights Reserved. Printed in the United States of America. This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by

any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit www.pearsoned.com/permissions/.

Text Credit: Page 23 Declaration of Independence: A Transcription, The U.S. National Archives and Records Administration.

PEARSON, ALWAYS LEARNING, and MYLAB are exclusive trademarks owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Unless otherwise indicated herein, any third-party trademarks that may appear in this work are the property of their respective owners and any references to third-party trademarks, logos or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc. or its affiliates, authors, licensees or distributors.

Library of Congress Cataloging-in-Publication Data

Names: Trappe, Wade, author. | Washington, Lawrence C., author.

Title: Introduction to cryptography : with coding theory / Wade Trappe, Lawrence Washington.

Description: 3rd edition. | [Hoboken, New Jersey] : [Pearson Education], [2020] | Includes bibliographical references and index. | Summary: "This book is based on a course in cryptography at the upper-level undergraduate and beginning graduate level that has

been given at the University of Maryland since 1997, and a course that has been taught at Rutgers University since 2003"— Provided by publisher.

Identifiers: LCCN 2019029691 | ISBN 9780134860992
(paperback)

Subjects: LCSH: Coding theory. | Cryptography.

Classification: LCC QA268.T73 2020 | DDC 005.8/24—dc23

LC record available at <https://lccn.loc.gov/2019029691>

ScoutAutomatedPrintCode

ISBN-13: 978-0-13-485906-4

ISBN-10: 0-13-485906-5



Contents

1. Preface	ix
1. 1 Overview of Cryptography and Its Applications	1
1. 1.1 Secure Communications	2
1. 1.2 Cryptographic Applications	8
2. 2 Classical Cryptosystems	10
1. 2.1 Shift Ciphers	11
1. 2.2 Affine Ciphers	12
1. 2.3 The Vigenère Cipher	14
1. 2.4 Substitution Ciphers	20
1. 2.5 Sherlock Holmes	23
1. 2.6 The Playfair and ADFGX Ciphers	26
1. 2.7 Enigma	29
1. 2.8 Exercises	33
1. 2.9 Computer Problems	37
3. 3 Basic Number Theory	40
1. 3.1 Basic Notions	40
1. 3.2 The Extended Euclidean Algorithm	44
1. 3.3 Congruences	47
1. 3.4 The Chinese Remainder Theorem	52
1. 3.5 Modular Exponentiation	54
1. 3.6 Fermat's Theorem and Euler's Theorem	55
1. 3.7 Primitive Roots	59
1. 3.8 Inverting Matrices Mod n	61
1. 3.9 Square Roots Mod n	62

- 10. 3.10 Legendre and Jacobi Symbols 64
 - 11. 3.11 Finite Fields 69
 - 12. 3.12 Continued Fractions 76
 - 13. 3.13 Exercises 78
 - 14. 3.14 Computer Problems 86
4. 4 The One-Time Pad 88
- 1. 4.1 Binary Numbers and ASCII 88
 - 2. 4.2 One-Time Pads 89
 - 3. 4.3 Multiple Use of a One-Time Pad 91
 - 4. 4.4 Perfect Secrecy of the One-Time Pad 94
 - 5. 4.5 Indistinguishability and Security 97
 - 6. 4.6 Exercises 100
5. 5 Stream Ciphers 104
- 1. 5.1 Pseudorandom Bit Generation 105
 - 2. 5.2 LFSR Sequences 107
 - 3. 5.3 RC4 113
 - 4. 5.4 Exercises 114
 - 5. 5.5 Computer Problems 117
6. 6 Block Ciphers 118
- 1. 6.1 Block Ciphers 118
 - 2. 6.2 Hill Ciphers 119
 - 3. 6.3 Modes of Operation 122
 - 4. 6.4 Multiple Encryption 129
 - 5. 6.5 Meet-in-the-Middle Attacks 130
 - 6. 6.6 Exercises 131
 - 7. 6.7 Computer Problems 135
7. 7 The Data Encryption Standard 136

1.	<u>7.1 Introduction</u>	136
2.	<u>7.2 A Simplified DES-Type Algorithm</u>	137
3.	<u>7.3 Differential Cryptanalysis</u>	140
4.	<u>7.4 DES</u>	145
5.	<u>7.5 Breaking DES</u>	152
6.	<u>7.6 Password Security</u>	155
7.	<u>7.7 Exercises</u>	157
8.	<u>7.8 Computer Problems</u>	159
8.	8 The Advanced Encryption Standard: Rijndael	160
1.	<u>8.1 The Basic Algorithm</u>	160
2.	<u>8.2 The Layers</u>	161
3.	<u>8.3 Decryption</u>	166
4.	<u>8.4 Design Considerations</u>	168
5.	<u>8.5 Exercises</u>	169
9.	9 The RSA Algorithm	171
1.	<u>9.1 The RSA Algorithm</u>	171
2.	<u>9.2 Attacks on RSA</u>	177
3.	<u>9.3 Primality Testing</u>	183
4.	<u>9.4 Factoring</u>	188
5.	<u>9.5 The RSA Challenge</u>	192
6.	<u>9.6 An Application to Treaty Verification</u>	194
7.	<u>9.7 The Public Key Concept</u>	195
8.	<u>9.8 Exercises</u>	197
9.	<u>9.9 Computer Problems</u>	207
10.	10 Discrete Logarithms	211
1.	<u>10.1 Discrete Logarithms</u>	211
2.	<u>10.2 Computing Discrete Logs</u>	212
3.	<u>10.3 Bit Commitment</u>	218

- 4. [10.4 Diffie-Hellman Key Exchange](#) 219
 - 5. [10.5 The ElGamal Public Key Cryptosystem](#) 221
 - 6. [10.6 Exercises](#) 223
 - 7. [10.7 Computer Problems](#) 225
11. [11 Hash Functions](#) 226
- 1. [11.1 Hash Functions](#) 226
 - 2. [11.2 Simple Hash Examples](#) 230
 - 3. [11.3 The Merkle-Damgård Construction](#) 231
 - 4. [11.4 SHA-2](#) 233
 - 5. [11.5 SHA-3/Keccak](#) 237
 - 6. [11.6 Exercises](#) 242
12. [12 Hash Functions: Attacks and Applications](#) 246
- 1. [12.1 Birthday Attacks](#) 246
 - 2. [12.2 Multicollisions](#) 249
 - 3. [12.3 The Random Oracle Model](#) 251
 - 4. [12.4 Using Hash Functions to Encrypt](#) 253
 - 5. [12.5 Message Authentication Codes](#) 255
 - 6. [12.6 Password Protocols](#) 256
 - 7. [12.7 Blockchains](#) 262
 - 8. [12.8 Exercises](#) 264
 - 9. [12.9 Computer Problems](#) 268
13. [13 Digital Signatures](#) 269
- 1. [13.1 RSA Signatures](#) 270
 - 2. [13.2 The ElGamal Signature Scheme](#) 271
 - 3. [13.3 Hashing and Signing](#) 273
 - 4. [13.4 Birthday Attacks on Signatures](#) 274
 - 5. [13.5 The Digital Signature Algorithm](#) 275
 - 6. [13.6 Exercises](#) 276

7. 13.7 Computer Problems	281
14. 14 What Can Go Wrong	282
1. 14.1 An Enigma “Feature”	282
2. 14.2 Choosing Primes for RSA	283
3. 14.3 WEP	284
4. 14.4 Exercises	288
15. 15 Security Protocols	290
1. 15.1 Intruders-in-the-Middle and Impostors	290
2. 15.2 Key Distribution	293
3. 15.3 Kerberos	299
4. 15.4 Public Key Infrastructures (PKI)	303
5. 15.5 X.509 Certificates	304
6. 15.6 Pretty Good Privacy	309
7. 15.7 SSL and TLS	312
8. 15.8 Secure Electronic Transaction	314
9. 15.9 Exercises	316
16. 16 Digital Cash	318
1. 16.1 Setting the Stage for Digital Economies	319
2. 16.2 A Digital Cash System	320
3. 16.3 Bitcoin Overview	326
4. 16.4 Cryptocurrencies	329
5. 16.5 Exercises	338
17. 17 Secret Sharing Schemes	340
1. 17.1 Secret Splitting	340
2. 17.2 Threshold Schemes	341
3. 17.3 Exercises	346
4. 17.4 Computer Problems	348

18. 18 Games 349

1. 18.1 Flipping Coins over the Telephone 349
2. 18.2 Poker over the Telephone 351
3. 18.3 Exercises 355

19. 19 Zero-Knowledge Techniques 357

1. 19.1 The Basic Setup 357
2. 19.2 The Feige-Fiat-Shamir Identification Scheme 359
3. 19.3 Exercises 361

20. 20 Information Theory 365

1. 20.1 Probability Review 365
2. 20.2 Entropy 367
3. 20.3 Huffman Codes 371
4. 20.4 Perfect Secrecy 373
5. 20.5 The Entropy of English 376
6. 20.6 Exercises 380

21. 21 Elliptic Curves 384

1. 21.1 The Addition Law 384
2. 21.2 Elliptic Curves Mod p 389
3. 21.3 Factoring with Elliptic Curves 393
4. 21.4 Elliptic Curves in Characteristic 2 396
5. 21.5 Elliptic Curve Cryptosystems 399
6. 21.6 Exercises 402
7. 21.7 Computer Problems 407

22. 22 Pairing-Based Cryptography 409

1. 22.1 Bilinear Pairings 409
2. 22.2 The MOV Attack 410
3. 22.3 Tripartite Diffie-Hellman 411

- 4. [22.4 Identity-Based Encryption](#) 412
 - 5. [22.5 Signatures](#) 414
 - 6. [22.6 Keyword Search](#) 417
 - 7. [22.7 Exercises](#) 419
23. [23 Lattice Methods](#) 421
- 1. [23.1 Lattices](#) 421
 - 2. [23.2 Lattice Reduction](#) 422
 - 3. [23.3 An Attack on RSA](#) 426
 - 4. [23.4 NTRU](#) 429
 - 5. [23.5 Another Lattice-Based Cryptosystem](#) 433
 - 6. [23.6 Post-Quantum Cryptography?](#) 435
 - 7. [23.7 Exercises](#) 435
24. [24 Error Correcting Codes](#) 437
- 1. [24.1 Introduction](#) 437
 - 2. [24.2 Error Correcting Codes](#) 442
 - 3. [24.3 Bounds on General Codes](#) 446
 - 4. [24.4 Linear Codes](#) 451
 - 5. [24.5 Hamming Codes](#) 457
 - 6. [24.6 Golay Codes](#) 459
 - 7. [24.7 Cyclic Codes](#) 466
 - 8. [24.8 BCH Codes](#) 472
 - 9. [24.9 Reed-Solomon Codes](#) 479
 - 10. [24.10 The McEliece Cryptosystem](#) 480
 - 11. [24.11 Other Topics](#) 483
 - 12. [24.12 Exercises](#) 483
 - 13. [24.13 Computer Problems](#) 487
25. [25 Quantum Techniques in Cryptography](#) 488
- 1. [25.1 A Quantum Experiment](#) 488

- 2. [25.2 Quantum Key Distribution](#) 491
 - 3. [25.3 Shor's Algorithm](#) 493
 - 4. [25.4 Exercises](#) 502
-
- 1. [A Mathematica[®] Examples](#) 503
 - 1. [A.1 Getting Started with Mathematica](#) 503
 - 2. [A.2 Some Commands](#) 504
 - 3. [A.3 Examples for Chapter 2](#) 505
 - 4. [A.4 Examples for Chapter 3](#) 508
 - 5. [A.5 Examples for Chapter 5](#) 511
 - 6. [A.6 Examples for Chapter 6](#) 513
 - 7. [A.7 Examples for Chapter 9](#) 514
 - 8. [A.8 Examples for Chapter 10](#) 520
 - 9. [A.9 Examples for Chapter 12](#) 521
 - 10. [A.10 Examples for Chapter 17](#) 521
 - 11. [A.11 Examples for Chapter 18](#) 522
 - 12. [A.12 Examples for Chapter 21](#) 523
 - 2. [B Maple[®] Examples](#) 527
 - 1. [B.1 Getting Started with Maple](#) 527
 - 2. [B.2 Some Commands](#) 528
 - 3. [B.3 Examples for Chapter 2](#) 529
 - 4. [B.4 Examples for Chapter 3](#) 533
 - 5. [B.5 Examples for Chapter 5](#) 536
 - 6. [B.6 Examples for Chapter 6](#) 538
 - 7. [B.7 Examples for Chapter 9](#) 539
 - 8. [B.8 Examples for Chapter 10](#) 546
 - 9. [B.9 Examples for Chapter 12](#) 547
 - 10. [B.10 Examples for Chapter 17](#) 548
 - 11. [B.11 Examples for Chapter 18](#) 549

12. B.12 Examples for Chapter 21 551

3. C MATLAB[®] Examples 555

1. C.1 Getting Started with MATLAB 556

2. C.2 Examples for Chapter 2 560

3. C.3 Examples for Chapter 3 566

4. C.4 Examples for Chapter 5 569

5. C.5 Examples for Chapter 6 571

6. C.6 Examples for Chapter 9 573

7. C.7 Examples for Chapter 10 581

8. C.8 Examples for Chapter 12 581

9. C.9 Examples for Chapter 17 582

10. C.10 Examples for Chapter 18 582

11. C.11 Examples for Chapter 21 585

4. D Sage Examples 591

1. D.1 Computations for Chapter 2 591

2. D.2 Computations for Chapter 3 594

3. D.3 Computations for Chapter 5 595

4. D.4 Computations for Chapter 6 596

5. D.5 Computations for Chapter 9 596

6. D.6 Computations for Chapter 10 597

7. D.7 Computations for Chapter 12 598

8. D.8 Computations for Chapter 17 598

9. D.9 Computations for Chapter 18 598

10. D.10 Computations for Chapter 21 599

5. E Answers and Hints for Selected Odd-Numbered Exercises 601

6. F Suggestions for Further Reading 607

7. Bibliography 608

8. Index 615

Preface

This book is based on a course in cryptography at the upper-level undergraduate and beginning graduate level that has been given at the University of Maryland since 1997, and a course that has been taught at Rutgers University since 2003. When designing the courses, we decided on the following requirements:

- The courses should be up-to-date and cover a broad selection of topics from a mathematical point of view.
- The material should be accessible to mathematically mature students having little background in number theory and computer programming.
- There should be examples involving numbers large enough to demonstrate how the algorithms really work.

We wanted to avoid concentrating solely on RSA and discrete logarithms, which would have made the courses mostly about number theory. We also did not want to focus on protocols and how to hack into friends' computers. That would have made the courses less mathematical than desired.

There are numerous topics in cryptology that can be discussed in an introductory course. We have tried to include many of them. The chapters represent, for the most part, topics that were covered during the different semesters we taught the course. There is certainly more material here than could be treated in most one-semester courses. The first thirteen chapters represent the core of the material. The choice of which of the remaining chapters are used depends on the level of the students and the objectives of the lecturer.

The chapters are numbered, thus giving them an ordering. However, except for Chapter 3 on number

theory, which pervades the subject, the chapters are fairly independent of each other and can be covered in almost any reasonable order. Since students have varied backgrounds in number theory, we have collected the basic number theory facts together in [Chapter 3](#) for ease of reference; however, we recommend introducing these concepts gradually throughout the course as they are needed.

The chapters on information theory, elliptic curves, quantum cryptography, lattice methods, and error correcting codes are somewhat more mathematical than the others. The chapter on error correcting codes was included, at the suggestion of several reviewers, because courses that include introductions to both cryptology and coding theory are fairly common.

Computer Examples

Suppose you want to give an example for RSA. You could choose two one-digit primes and pretend to be working with fifty-digit primes, or you could use your favorite software package to do an actual example with large primes. Or perhaps you are working with shift ciphers and are trying to decrypt a message by trying all 26 shifts of the ciphertext. This should also be done on a computer.

Additionally, at the end of the book are appendices containing computer examples written in each of Mathematica[®], Maple[®], MATLAB[®], and Sage that show how to do such calculations. These languages were chosen because they are user friendly and do not require prior programming experience. Although the course has been taught successfully without computers, these examples are an integral part of the book and should be studied, if at all possible. Not only do they contain numerical examples of how to do certain computations

but also they demonstrate important ideas and issues that arise. They were placed at the end of the book because of the logistic and aesthetic problems of including extensive computer examples in these languages at the ends of chapters.

Additionally, programs available in Mathematica, Maple, and MATLAB can be downloaded from the Web site (bit.ly/2JbcS6p). Homework problems (the computer problems in various chapters) based on the software allow students to play with examples individually. Of course, students having more programming background could write their own programs instead. In a classroom, all that is needed is a computer (with one of the languages installed) and a projector in order to produce meaningful examples as the lecture is being given.

New to the Third Edition

Two major changes have informed this edition: Changes to the field of cryptography and a change in the format of the text. We address these issues separately, although there is an interplay between the two:

Content Changes

Cryptography is a quickly changing field. We have made many changes to the text since the last edition:

- Reorganized content previously in two chapters to four separate chapters on Stream Ciphers (including RC4), Block Ciphers, DES and AES (Chapters 5–8, respectively). The RC4 material, in particular, is new.
- Heavily revised the chapters on hash functions. Chapter 11 (Hash functions) now includes sections on SHA-2 and SHA-3. Chapter 12 (Hash functions: Attacks and Applications) now includes material on message authentication codes, password protocols, and blockchains.

- The short section on the one-time pad has been expanded to become [Chapter 4](#), which includes sections on multiple use of the one-time pad, perfect secrecy, and ciphertext indistinguishability.
- Added [Chapter 14](#), “What Can Go Wrong,” which shows what can happen when cryptographic algorithms are used or designed incorrectly.
- Expanded [Chapter 16](#) on digital cash to include Bitcoin and cryptocurrencies.
- Added [Chapter 22](#), which gives an introduction to Pairing-Based Cryptography.
- Updated the exposition throughout the book to reflect recent developments.
- Added references to the Maple, Mathematica, MATLAB, and Sage appendices in relevant locations in the text.
- Added many new exercises.
- Added a section at the back of the book that contains answers or hints to a majority of the odd-numbered problems.

Format Changes

A focus of this revision was transforming the text from a print-based learning tool to a digital learning tool. The eText is therefore filled with content and tools that will help bring the content of the course to life for students in new ways and help improve instruction. Specifically, the following are features that are available only in the eText:

- Interactive Examples. We have added a number of opportunities for students to interact with content in a dynamic manner in order to build or enhance understanding. Interactive examples allow students to explore concepts in ways that are not possible without technology.
- Quick Questions. These questions, built into the narrative, provide opportunities for students to check and clarify understanding. Some help address potential misconceptions.
- Notes, Labels, and Highlights. Notes can be added to the eText by instructors. These notes are visible to all students in the course, allowing instructors to add their personal observations or directions to important topics, call out need-to-know information, or clarify difficult concepts. Students can add their own notes,

labels, and highlights to the eText, helping them focus on what they need to study. The customizable Notebook allows students to filter, arrange, and group their notes in a way that makes sense to them.

- Dashboard. Instructors can create reading assignments and see the time spent in the eText so that they can plan more effective instruction.
- Portability. Portable access lets students read their eText whenever they have a moment in their day, on Android and iOS mobile phones and tablets. Even without an Internet connection, offline reading ensures students never miss a chance to learn.
- Ease-of-Use. Straightforward setup makes it easy for instructors to get their class up and reading quickly on the first day of class. In addition, Learning Management System (LMS) integration provides institutions, instructors, and students with single sign-on access to the eText via many popular LMSs.
- Supplements. An Instructors' Solutions Manual can be downloaded by qualified instructors from the textbook's webpage at www.pearson.com.

Acknowledgments

Many people helped and provided encouragement during the preparation of this book. First, we would like to thank our students, whose enthusiasm, insights, and suggestions contributed greatly. We are especially grateful to many people who have provided corrections and other input, especially Bill Gasarch, Jeff Adams, Jonathan Rosenberg, and Tim Strobell. We would like to thank Wenyuan Xu, Qing Li, and Pandurang Kamat, who drew several of the diagrams and provided feedback on the new material for the second edition. We have enjoyed working with the staff at Pearson, especially Jeff Weidenaar and Tara Corpuz.

The reviewers deserve special thanks: their suggestions on the exposition and the organization of the topics greatly enhanced the final result. The reviewers marked with an asterisk (*) provided input for this edition.

• * Anurag Agarwal, Rochester Institute of Technology

- * Pradeep Atrey, University at Albany
 - Eric Bach, University of Wisconsin
 - James W. Brewer, Florida Atlantic University
 - Thomas P. Cahill, NYU
 - Agnes Chan, Northeastern University
 - * Nathan Chenette, Rose-Hulman Institute of Technology
 - * Claude Crépeau, McGill University
 - * Reza Curtmola, New Jersey Institute of Technology
 - * Ahmed Desoky, University of Louisville
- Anthony Ephremides, University of Maryland, College Park
- * David J. Fawcett, Lawrence Tech University
 - * Jason Gibson, Eastern Kentucky University
 - * K. Gopalakrishnan, East Carolina University
- David Grant, University of Colorado, Boulder
- Jugal K. Kalita, University of Colorado, Colorado Springs
- * Saroja Kanchi, Kettering University
 - * Andrew Klapper, University of Kentucky
 - * Amanda Knecht, Villanova University
- Edmund Lamagna, University of Rhode Island
- * Aihua Li, Montclair State University
 - * Spyros S. Magliveras, Florida Atlantic University
 - * Nathan McNew, Towson University
 - * Nick Novotny, IUPUI
- David M. Pozar, University of Massachusetts, Amherst
- * Emma Previato, Boston University
 - * Hamzeh Roumani, York University
 - * Bonnie Saunders, University of Illinois, Chicago
 - * Ravi Shankar, University of Oklahoma
 - * Ernie Stitzinger, North Carolina State

- * Armin Straub, University of South Alabama
- J. Felipe Voloch, University of Texas, Austin
- Daniel F. Warren, Naval Postgraduate School
- * Simon Whitehouse, Alfred State College
- Siman Wong, University of Massachusetts, Amherst
- * Huapeng Wu, University of Windsor

Wade thanks Nisha Gilra, who provided encouragement and advice; Sheilagh O'Hare for introducing him to the field of cryptography; and K. J. Ray Liu for his support. Larry thanks Susan Zengerle and Patrick Washington for their patience, help, and encouragement during the writing of this book.

Of course, we welcome suggestions and corrections. An errata page can be found at (bit.ly/2J8nN0w) or at the link on the book's general Web site (bit.ly/2T544yu).

Wade Trappe

trappe@winlab.rutgers.edu

Lawrence C. Washington

lcw@math.umd.edu

Chapter 1 Overview of Cryptography and Its Applications

People have always had a fascination with keeping information away from others. As children, many of us had magic decoder rings for exchanging coded messages with our friends and possibly keeping secrets from parents, siblings, or teachers. History is filled with examples where people tried to keep information secret from adversaries. Kings and generals communicated with their troops using basic cryptographic methods to prevent the enemy from learning sensitive military information. In fact, Julius Caesar reportedly used a simple cipher, which has been named after him.

As society has evolved, the need for more sophisticated methods of protecting data has increased. Now, with the information era at hand, the need is more pronounced than ever. As the world becomes more connected, the demand for information and electronic services is growing, and with the increased demand comes increased dependency on electronic systems. Already the exchange of sensitive information, such as credit card numbers, over the Internet is common practice. Protecting data and electronic systems is crucial to our way of living.

The techniques needed to protect data belong to the field of cryptography. Actually, the subject has three names, **cryptography**, **cryptology**, and **cryptanalysis**, which are often used interchangeably. Technically, however, cryptology is the all-inclusive term for the study of communication over nonsecure channels, and related problems. The process of designing systems to do this is

called cryptography. Cryptanalysis deals with breaking such systems. Of course, it is essentially impossible to do either cryptography or cryptanalysis without having a good understanding of the methods of both areas.

Often the term **coding theory** is used to describe cryptography; however, this can lead to confusion. Coding theory deals with representing input information symbols by output symbols called code symbols. There are three basic applications that coding theory covers: compression, secrecy, and error correction. Over the past few decades, the term coding theory has become associated predominantly with error correcting codes. Coding theory thus studies communication over noisy channels and how to ensure that the message received is the correct message, as opposed to cryptography, which protects communication over nonsecure channels.

Although error correcting codes are only a secondary focus of this book, we should emphasize that, in any real-world system, error correcting codes are used in conjunction with encryption, since the change of a single bit is enough to destroy the message completely in a well-designed cryptosystem.

Modern cryptography is a field that draws heavily upon mathematics, computer science, and cleverness. This book provides an introduction to the mathematics and protocols needed to make data transmission and electronic systems secure, along with techniques such as electronic signatures and secret sharing.

1.1 Secure Communications

In the basic communication scenario, depicted in Figure 1.1, there are two parties, we'll call them Alice and Bob, who want to communicate with each other. A third party, Eve, is a potential eavesdropper.

Figure 1.1 The Basic Communication Scenario for Cryptography.

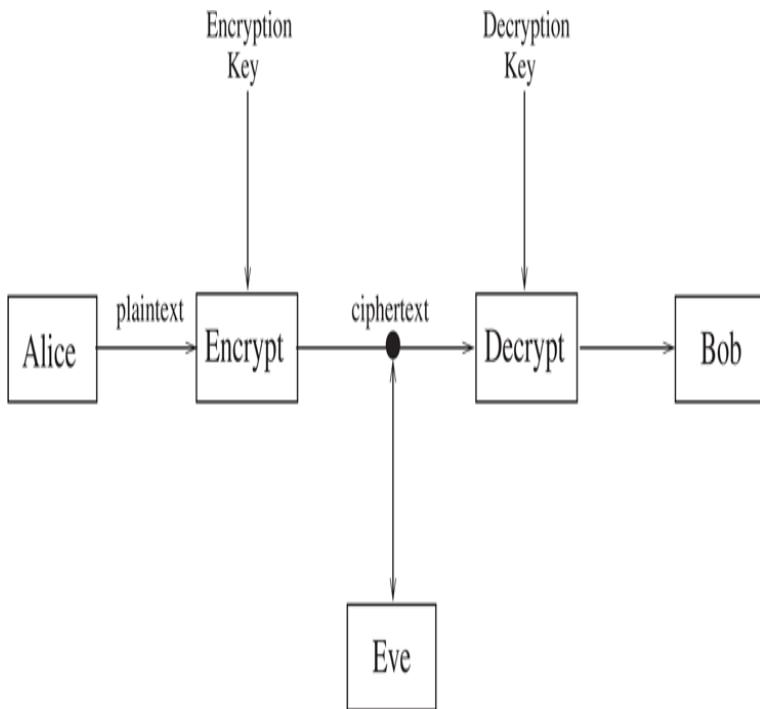


Figure 1.1 Full Alternative Text

When Alice wants to send a message, called the **plaintext**, to Bob, she encrypts it using a method prearranged with Bob. Usually, the encryption method is assumed to be known to Eve; what keeps the message secret is a **key**. When Bob receives the encrypted

message, called the **ciphertext**, he changes it back to the plaintext using a decryption key.

Eve could have one of the following goals:

1. Read the message.
2. Find the key and thus read all messages encrypted with that key.
3. Corrupt Alice's message into another message in such a way that Bob will think Alice sent the altered message.
4. Masquerade as Alice, and thus communicate with Bob even though Bob believes he is communicating with Alice.

Which case we're in depends on how evil Eve is. Cases (3) and (4) relate to issues of integrity and authentication, respectively. We'll discuss these shortly. A more active and malicious adversary, corresponding to cases (3) and (4), is sometimes called Mallory in the literature. More passive observers (as in cases (1) and (2)) are sometimes named Oscar. We'll generally use only Eve, and assume she is as bad as the situation allows.

1.1.1 Possible Attacks

There are four main types of attack that Eve might be able to use. The differences among these types of attacks are the amounts of information Eve has available to her when trying to determine the key. The four attacks are as follows:

1. Ciphertext only: Eve has only a copy of the ciphertext.
2. Known plaintext: Eve has a copy of a ciphertext and the corresponding plaintext. For example, suppose Eve intercepts an encrypted press release, then sees the decrypted release the next day. If she can deduce the decryption key, and if Alice doesn't change the key, Eve can read all future messages. Or, if Alice always starts her messages with "Dear Bob," then Eve has a small piece of ciphertext and corresponding plaintext. For many weak cryptosystems, this suffices to find the key. Even for stronger

systems such as the German Enigma machine used in World War II, this amount of information has been useful.

3. Chosen plaintext: Eve gains temporary access to the encryption machine. She cannot open it to find the key; however, she can encrypt a large number of suitably chosen plaintexts and try to use the resulting ciphertexts to deduce the key.
4. Chosen ciphertext: Eve obtains temporary access to the decryption machine, uses it to “decrypt” several strings of symbols, and tries to use the results to deduce the key.

A chosen plaintext attack could happen as follows. You want to identify an airplane as friend or foe. Send a random message to the plane, which encrypts the message automatically and sends it back. Only a friendly airplane is assumed to have the correct key. Compare the message from the plane with the correctly encrypted message. If they match, the plane is friendly. If not, it's the enemy. However, the enemy can send a large number of chosen messages to one of your planes and look at the resulting ciphertexts. If this allows them to deduce the key, the enemy can equip their planes so they can masquerade as friendly.

An example of a known plaintext attack reportedly happened in World War II in the Sahara Desert. An isolated German outpost every day sent an identical message saying that there was nothing new to report, but of course it was encrypted with the key being used that day. So each day the Allies had a plaintext-ciphertext pair that was extremely useful in determining the key. In fact, during the Sahara campaign, General Montgomery was carefully directed around the outpost so that the transmissions would not be stopped.

One of the most important assumptions in modern cryptography is **Kerckhoffs's principle**: In assessing the security of a cryptosystem, one should always assume the enemy knows the method being used. This principle was enunciated by Auguste Kerckhoffs in 1883 in his classic treatise *La Cryptographie Militaire*. The enemy

can obtain this information in many ways. For example, encryption/decryption machines can be captured and analyzed. Or people can defect or be captured. The security of the system should therefore be based on the key and not on the obscurity of the algorithm used. Consequently, we always assume that Eve has knowledge of the algorithm that is used to perform encryption.

1.1.2 Symmetric and Public Key Algorithms

Encryption/decryption methods fall into two categories: **symmetric key** and **public key**. In symmetric key algorithms, the encryption and decryption keys are known to both Alice and Bob. For example, the encryption key is shared and the decryption key is easily calculated from it. In many cases, the encryption key and the decryption key are the same. All of the classical (pre-1970) cryptosystems are symmetric, as are the more recent Data Encryption Standard (DES) and Advanced Encryption Standard (AES).

Public key algorithms were introduced in the 1970s and revolutionized cryptography. Suppose Alice wants to communicate securely with Bob, but they are hundreds of kilometers apart and have not agreed on a key to use. It seems almost impossible for them to do this without first getting together to agree on a key, or using a trusted courier to carry the key from one to the other. Certainly Alice cannot send a message over open channels to tell Bob the key, and then send the ciphertext encrypted with this key. The amazing fact is that this problem has a solution, called public key cryptography. The encryption key is made public, but it is computationally infeasible to find the decryption key without information known only to Bob. The most popular implementation is RSA (see Chapter 9), which is based on the difficulty of factoring

large integers. Other versions (see [Chapters 10, 23, and 24](#)) are the ElGamal system (based on the discrete log problem), NTRU (lattice based) and the McEliece system (based on error correcting codes).

Here is a nonmathematical way to do public key communication. Bob sends Alice a box and an unlocked padlock. Alice puts her message in the box, locks Bob's lock on it, and sends the box back to Bob. Of course, only Bob can open the box and read the message. The public key methods mentioned previously are mathematical realizations of this idea. Clearly there are questions of authentication that must be dealt with. For example, Eve could intercept the first transmission and substitute her own lock. If she then intercepts the locked box when Alice sends it back to Bob, Eve can unlock her lock and read Alice's message. This is a general problem that must be addressed with any such system.

Public key cryptography represents what is possibly the final step in an interesting historical progression. In the earliest years of cryptography, security depended on keeping the encryption method secret. Later, the method was assumed known, and the security depended on keeping the (symmetric) key private or unknown to adversaries. In public key cryptography, the method and the encryption key are made public, and everyone knows what must be done to find the decryption key. The security rests on the fact (or hope) that this is computationally infeasible. It's rather paradoxical that an increase in the power of cryptographic algorithms over the years has corresponded to an increase in the amount of information given to an adversary about such algorithms.

Public key methods are very powerful, and it might seem that they make the use of symmetric key cryptography obsolete. However, this added flexibility is not free and comes at a computational cost. The amount of

computation needed in public key algorithms is typically several orders of magnitude more than the amount of computation needed in algorithms such as DES or AES/Rijndael. The rule of thumb is that public key methods should not be used for encrypting large quantities of data. For this reason, public key methods are used in applications where only small amounts of data must be processed (for example, digital signatures and sending keys to be used in symmetric key algorithms).

Within symmetric key cryptography, there are two types of ciphers: stream ciphers and block ciphers. In stream ciphers, the data are fed into the algorithm in small pieces (bits or characters), and the output is produced in corresponding small pieces. We discuss stream ciphers in [Chapter 5](#). In block ciphers, however, a block of input bits is collected and fed into the algorithm all at once, and the output is a block of bits. Mostly we shall be concerned with block ciphers. In particular, we cover two very significant examples. The first is DES, and the second is AES, which was selected in the year 2000 by the National Institute for Standards and Technology as the replacement for DES. Public key methods such as RSA can also be regarded as block ciphers.

Finally, we mention a historical distinction between different types of encryption, namely **codes** and **ciphers**. In a code, words or certain letter combinations are replaced by codewords (which may be strings of symbols). For example, the British navy in World War I used 03680C, 36276C, and 50302C to represent *shipped at*, *shipped by*, and *shipped from*, respectively. Codes have the disadvantage that unanticipated words cannot be used. A cipher, on the other hand, does not use the linguistic structure of the message but rather encrypts every string of characters, meaningful or not, by some algorithm. A cipher is therefore more versatile than a code. In the early days of cryptography, codes were

commonly used, sometimes in conjunction with ciphers. They are still used today; covert operations are often given code names. However, any secret that is to remain secure needs to be encrypted with a cipher. In this book, we'll deal exclusively with ciphers.

1.1.3 Key Length

The security of cryptographic algorithms is a difficult property to measure. Most algorithms employ keys, and the security of the algorithm is related to how difficult it is for an adversary to determine the key. The most obvious approach is to try every possible key and see which ones yield meaningful decryptions. Such an attack is called a **brute force attack**. In a brute force attack, the length of the key is directly related to how long it will take to search the entire keyspace. For example, if a key is 16 bits long, then there are

$2^{16} = 65536$ possible keys. The DES algorithm has a 56-bit key and thus has

$2^{56} \approx 7.2 \times 10^{16}$ possible keys.

In many situations we'll encounter in this book, it will seem that a system can be broken by simply trying all possible keys. However, this is often easier said than done. Suppose you need to try 10^{30} possibilities and you have a computer that can do 10^9 such calculations each second. There are around 3×10^7 seconds in a year, so it would take a little more than 3×10^{13} years to complete the task, longer than the predicted life of the universe.

Longer keys are advantageous but are not guaranteed to make an adversary's task difficult. The algorithm itself also plays a critical role. Some algorithms might be able to be attacked by means other than brute force, and some algorithms just don't make very efficient use of their

keys' bits. This is a very important point to keep in mind.
Not all 128-bit algorithms are created equal!

For example, one of the easiest cryptosystems to break is the substitution cipher, which we discuss in [Section 2.4](#). The number of possible keys is $26! \approx 4 \times 10^{26}$. In contrast, DES (see [Chapter 7](#)) has only $2^{56} \approx 7.2 \times 10^{16}$ keys. But it typically takes over a day on a specially designed computer to find a DES key. The difference is that an attack on a substitution cipher uses the underlying structure of the language, while the attack on DES is by brute force, trying all possible keys.

A brute force attack should be the last resort. A cryptanalyst always hopes to find an attack that is faster. Examples we'll meet are frequency analysis (for the substitution and Vigenère ciphers) and birthday attacks (for discrete logs).

We also warn the reader that just because an algorithm seems secure now, we can't assume that it will remain so. Human ingenuity has led to creative attacks on cryptographic protocols. There are many examples in modern cryptography where an algorithm or protocol was successfully attacked because of a loophole presented by poor implementation, or just because of advances in technology. The DES algorithm, which withstood 20 years of cryptographic scrutiny, ultimately succumbed to attacks by a well-designed parallel computer. Even as you read this book, research in quantum computing is underway, which could dramatically alter the terrain of future cryptographic algorithms.

For example, the security of several systems we'll study depends on the difficulty of factoring large integers, say of around 600 digits. Suppose you want to factor a number n of this size. The method used in elementary school is to divide n by all of the primes up to the square

root of n . There are approximately 1.4×10^{297} primes less than 10^{300} . Trying each one is impossible. The number of electrons in the universe is estimated to be less than 10^{90} . Long before you finish your calculation, you'll get a call from the electric company asking you to stop. Clearly, more sophisticated factoring algorithms must be used, rather than this brute force type of attack. When RSA was invented, there were some good factoring algorithms available, but it was predicted that a 129-digit number such as the RSA challenge number (see Chapter 9) would not be factored within the foreseeable future. However, advances in algorithms and computer architecture have made such factorizations fairly routine (although they still require substantial computing resources), so now numbers of several hundred digits are recommended for security. But if a full-scale quantum computer is ever built, factorizations of even these numbers will be easy, and the whole RSA scheme (along with many other methods) will need to be reconsidered.

A natural question, therefore, is whether there are any unbreakable cryptosystems, and, if so, why aren't they used all the time?

The answer is yes; there is a system, known as the one-time pad, that is unbreakable. Even a brute force attack will not yield the key. But the unfortunate truth is that the expense of using a one-time pad is enormous. It requires exchanging a key that is as long as the plaintext, and even then the key can only be used once. Therefore, one opts for algorithms that, when implemented correctly with the appropriate key size, are unbreakable in any reasonable amount of time.

An important point when considering key size is that, in many cases, one can mathematically increase security by a slight increase in key size, but this is not always practical. If you are working with chips that can handle words of 64 bits, then an increase in the key size from 64

to 65 bits could mean redesigning your hardware, which could be expensive. Therefore, designing good cryptosystems involves both mathematical and engineering considerations.

Finally, we need a few words about the size of numbers. Your intuition might say that working with a 20-digit number takes twice as long as working with a 10-digit number. That is true in some algorithms. However, if you count up to 10^{10} , you are not even close to 10^{20} ; you are only one 10 billionth of the way there. Similarly, a brute force attack against a 60-bit key takes a billion times longer than one against a 30-bit key.

There are two ways to measure the size of numbers: the actual magnitude of the number n , and the number of digits in its decimal representation (we could also use its binary representation), which is approximately $\log_{10}(n)$. The number of single-digit multiplications needed to square a k -digit number n , using the standard algorithm from elementary school, is k^2 , or approximately $(\log_{10} n)^2$. The number of divisions needed to factor a number n by dividing by all primes up to the square root of n is around $n^{1/2}$. An algorithm that runs in time a power of $\log n$ is much more desirable than one that runs in time a power of n . In the present example, if we double the number of digits in n , the time it takes to square n increases by a factor of 4, while the time it takes to factor n increases enormously. Of course, there are better algorithms available for both of these operations, but, at present, factorization takes significantly longer than multiplication.

We'll meet algorithms that take time a power of $\log n$ to perform certain calculations (for example, finding greatest common divisors and doing modular exponentiation). There are other computations for which the best known algorithms run only slightly better than a power of n (for example, factoring and finding discrete

logarithms). The interplay between the fast algorithms and the slower ones is the basis of several cryptographic algorithms that we'll encounter in this book.

1.2 Cryptographic Applications

Cryptography is not only about encrypting and decrypting messages, it is also about solving real-world problems that require information security. There are four main objectives that arise:

1. Confidentiality: Eve should not be able to read Alice's message to Bob. The main tools are encryption and decryption algorithms.
2. Data integrity: Bob wants to be sure that Alice's message has not been altered. For example, transmission errors might occur. Also, an adversary might intercept the transmission and alter it before it reaches the intended recipient. Many cryptographic primitives, such as hash functions, provide methods to detect data manipulation by malicious or accidental adversaries.
3. Authentication: Bob wants to be sure that only Alice could have sent the message he received. Under this heading, we also include identification schemes and password protocols (in which case, Bob is the computer). There are actually two types of authentication that arise in cryptography: entity authentication and data-origin authentication. Often the term *identification* is used to specify entity authentication, which is concerned with proving the identity of the parties involved in a communication. Data-origin authentication focuses on tying the information about the origin of the data, such as the creator and time of creation, with the data.
4. Non-repudiation: Alice cannot claim she did not send the message. Non-repudiation is particularly important in electronic commerce applications, where it is important that a consumer cannot deny the authorization of a purchase.

Authentication and non-repudiation are closely related concepts, but there is a difference. In a symmetric key cryptosystem, Bob can be sure that a message comes from Alice (or someone who knows Alice's key) since no one else could have encrypted the message that Bob decrypts successfully. Therefore, authentication is automatic. However, he cannot prove to anyone else that Alice sent the message, since he could have sent the

message himself. Therefore, non-repudiation is essentially impossible. In a public key cryptosystem, both authentication and non-repudiation can be achieved (see Chapters 9, 13, and 15).

Much of this book will present specific cryptographic applications, both in the text and as exercises. Here is an overview.

Digital signatures: One of the most important features of a paper and ink letter is the signature. When a document is signed, an individual's identity is tied to the message. The assumption is that it is difficult for another person to forge the signature onto another document. Electronic messages, however, are very easy to copy exactly. How do we prevent an adversary from cutting the signature off one document and attaching it to another electronic document? We shall study cryptographic protocols that allow for electronic messages to be signed in such a way that everyone believes that the signer was the person who signed the document, and such that the signer cannot deny signing the document.

Identification: When logging into a machine or initiating a communication link, a user needs to identify herself or himself. But simply typing in a user name is not sufficient as it does not prove that the user is really who he or she claims to be. Typically a password is used. We shall touch upon various methods for identifying oneself. In the chapter on DES we discuss password files. Later, we present the Feige-Fiat-Shamir identification scheme, which is a zero-knowledge method for proving identity without revealing a password.

Key establishment: When large quantities of data need to be encrypted, it is best to use symmetric key encryption algorithms. But how does Alice give the secret key to Bob when she doesn't have the opportunity to meet him personally? There are various ways to do this. One way

uses public key cryptography. Another method is the Diffie-Hellman key exchange algorithm. A different approach to this problem is to have a trusted third party give keys to Alice and Bob. Two examples are Blom's key generation scheme and Kerberos, which is a very popular symmetric cryptographic protocol that provides authentication and security in key exchange between users on a network.

Secret sharing: In Chapter 17, we introduce secret sharing schemes. Suppose that you have a combination to a bank safe, but you don't want to trust any single person with the combination to the safe. Rather, you would like to divide the combination among a group of people, so that at least two of these people must be present in order to open the safe. Secret sharing solves this problem.

Security protocols: How can we carry out secure transactions over open channels such as the Internet, and how can we protect credit card information from fraudulent merchants? We discuss various protocols, such as SSL and SET.

Electronic cash: Credit cards and similar devices are convenient but do not provide anonymity. Clearly a form of electronic cash could be useful, at least to some people. However, electronic entities can be copied. We give an example of an electronic cash system that provides anonymity but catches counterfeiters, and we discuss cryptocurrencies, especially Bitcoin.

Games: How can you flip coins or play poker with people who are not in the same room as you? Dealing the cards, for example, presents a problem. We show how cryptographic ideas can solve these problems.

Chapter 2 Classical Cryptosystems

Methods of making messages unintelligible to adversaries have been important throughout history. In this chapter we shall cover some of the older cryptosystems that were primarily used before the advent of the computer. These cryptosystems are too weak to be of much use today, especially with computers at our disposal, but they give good illustrations of several of the important ideas of cryptology.

First, for these simple cryptosystems, we make some conventions.

- *plaintext* will be written in lowercase letters and *CIPHERTEXT* will be written in capital letters (except in the computer problems).
- The letters of the alphabet are assigned numbers as follows:

<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>	<i>k</i>	<i>l</i>	<i>m</i>	<i>n</i>	<i>o</i>	<i>p</i>
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>q</i>	<i>r</i>	<i>s</i>	<i>t</i>	<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>						
16	17	18	19	20	21	22	23	24	25						

Note that we start with $a = 0$, so z is letter number 25. Because many people are accustomed to a being 1 and z being 26, the present convention can be annoying, but it is standard for the elementary cryptosystems that we'll consider.

- Spaces and punctuation are omitted. This is even more annoying, but it is almost always possible to replace the spaces in the plaintext after decrypting. If spaces were left in, there would be two choices. They could be left as spaces; but this yields so much information on the structure of the message that decryption becomes easier. Or they could be encrypted; but then they would dominate frequency counts (unless the message averages at least eight letters per word), again simplifying decryption.

Note: In this chapter, we'll be using some concepts from number theory, especially modular arithmetic. If you are

not familiar with congruences, you should read the first three sections of [Chapter 3](#) before proceeding.

2.1 Shift Ciphers

One of the earliest cryptosystems is often attributed to Julius Caesar. Suppose he wanted to send a plaintext such as

gaul is divided into three parts

but he didn't want Brutus to read it. He shifted each letter backwards by three places, so d became A , e became B , f became C , etc. The beginning of the alphabet wrapped around to the end, so a became X , b became Y , and c became Z . The ciphertext was then

DXRIFPAFSFABAFKQLQEOPBMXOQP.

Decryption was accomplished by shifting FORWARD by three spaces (and trying to figure out how to put the spaces back in).

We now give the general situation. *If you are not familiar with modular arithmetic, read the first few pages of Chapter 3 before continuing.*

Label the letters as integers from 0 to 25. The key is an integer κ with $0 \leq \kappa \leq 25$. The encryption process is

$$x \mapsto x + \kappa \pmod{26}.$$

Decryption is $x \mapsto x - \kappa \pmod{26}$. For example, Caesar used $\kappa = 23 \equiv -3$.

Let's see how the four types of attack work.

1. Ciphertext only: Eve has only the ciphertext. Her best strategy is an exhaustive search, since there are only 26 possible keys. See Example 1 in the Computer Appendices. If the message is longer than a few letters (we will make this more precise later when we discuss entropy), it is unlikely that there is more than one meaningful message that could be the plaintext. If you don't believe this, try to find some words of four or more letters that are

shifts of each other. Three such words are given in [Exercises 1](#) and [2](#). Another possible attack, if the message is sufficiently long, is to do a frequency count for the various letters. The letter e occurs most frequently in most English texts. Suppose the letter L appears most frequently in the ciphertext. Since $e = 4$ and $L = 11$, a reasonable guess is that $\kappa = 11 - 4 = 7$. However, for shift ciphers this method takes much longer than an exhaustive search, plus it requires many more letters in the message in order for it to work (anything short, such as this, might not contain a common symbol, thus changing statistical counts).

2. Known plaintext: If you know just one letter of the plaintext along with the corresponding letter of ciphertext, you can deduce the key. For example, if you know $t (= 19)$ encrypts to $D (= 3)$, then the key is $\kappa \equiv 3 - 19 \equiv -16 \equiv 10 \pmod{26}$.
3. Chosen plaintext: Choose the letter a as the plaintext. The ciphertext gives the key. For example, if the ciphertext is H , then the key is 7.
4. Chosen ciphertext: Choose the letter A as ciphertext. The plaintext is the negative of the key. For example, if the plaintext is h , the key is $-7 \equiv 19 \pmod{26}$.

2.2 Affine Ciphers

The shift ciphers may be generalized and slightly strengthened as follows. Choose two integers α and β , with $\gcd(\alpha, 26) = 1$, and consider the function (called an *affine function*)

$$x \mapsto \alpha x + \beta \pmod{26}.$$

For example, let $\alpha = 9$ and $\beta = 2$, so we are working with $9x + 2$. Take a plaintext letter such as $h (= 7)$. It is encrypted to $9 \cdot 7 + 2 \equiv 65 \equiv 13 \pmod{26}$, which is the letter N . Using the same function, we obtain

$$\text{affine} \mapsto CVVWPM.$$

How do we decrypt? If we were working with rational numbers rather than mod 26, we would start with

$y = 9x + 2$ and solve: $x = \frac{1}{9}(y - 2)$. But $\frac{1}{9}$ needs to be reinterpreted when we work mod 26. Since $\gcd(9, 26) = 1$, there is a multiplicative inverse for 9 (mod 26) (if this last sentence doesn't make sense to you, read [Section 3.3](#) now). In fact, $9 \cdot 3 \equiv 1 \pmod{26}$, so 3 is the desired inverse and can be used in place of $\frac{1}{9}$.

We therefore have

$$x \equiv 3(y - 2) \equiv 3y - 6 \equiv 3y + 20 \pmod{26}.$$

Let's try this. The letter $V (= 21)$ is mapped to $3 \cdot 21 + 20 \equiv 83 \equiv 5 \pmod{26}$, which is the letter f . Similarly, we see that the ciphertext $CVVWPM$ is decrypted back to *affine*. For more examples, see Examples 2 and 3 in the Computer Appendices.

Suppose we try to use the function $13x + 4$ as our encryption function. We obtain

$$\text{input} \mapsto ERER.$$

If we alter the input, we obtain

$$\text{alter} \mapsto \text{ERRER}.$$

Clearly this function leads to errors. It is impossible to decrypt, since several plaintexts yield the same ciphertext. In particular, we note that encryption must be one-to-one, and this fails in the present case.

What goes wrong in this example? If we solve

$y = 13x + 4$, we obtain $x = \frac{1}{13}(y - 4)$. But $\frac{1}{13}$ does not exist mod 26 since $\gcd(13, 26) = 13 \neq 1$. More generally, it can be shown that $\alpha x + \beta$ is a one-to-one function mod 26 if and only if $\gcd(\alpha, 26) = 1$. In this case, decryption uses $x \equiv \alpha^* y - \alpha^* \beta \pmod{26}$, where $\alpha \alpha^* \equiv 1 \pmod{26}$. So decryption is also accomplished by an affine function.

The key for this encryption method is the pair (α, β) . There are 12 possible choices for α with $\gcd(\alpha, 26) = 1$ and there are 26 choices for β (since we are working mod 26, we only need to consider α and β between 0 and 25). Therefore, there are $12 \cdot 26 = 312$ choices for the key.

Let's look at the possible attacks.

1. Ciphertext only: An exhaustive search through all 312 keys would take longer than the corresponding search in the case of the shift cipher; however, it would be very easy to do on a computer. When all possibilities for the key are tried, a fairly short ciphertext, say around 20 characters, will probably correspond to only one meaningful plaintext, thus allowing the determination of the key. It would also be possible to use frequency counts, though this would require much longer texts.
2. Known plaintext: With a little luck, knowing two letters of the plaintext and the corresponding letters of the ciphertext suffices to find the key. In any case, the number of possibilities for the key is greatly reduced and a few more letters should yield the key.

For example, suppose the plaintext starts with *if* and the corresponding ciphertext is *PQ*. In numbers, this means that 8 ($= i$) maps to 15 ($= P$) and 5 maps to 16. Therefore, we have the equations

$$8\alpha + \beta \equiv 15 \text{ and } 5\alpha + \beta \equiv 16 \pmod{26}.$$

Subtracting yields $3\alpha \equiv -1 \equiv 25 \pmod{26}$, which has the unique solution $\alpha = 17$. Using the first equation, we find $8 \cdot 17 + \beta \equiv 15 \pmod{26}$, which yields $\beta = 9$.

Suppose instead that the plaintext go corresponds to the ciphertext TH . We obtain the equations

$$6\alpha + \beta \equiv 19 \text{ and } 14\alpha + \beta \equiv 7 \pmod{26}.$$

Subtracting yields $-8\alpha \equiv 12 \pmod{26}$. Since $\gcd(-8, 26) = 2$, this has two solutions: $\alpha = 5, 18$. The corresponding values of β are both 15 (this is not a coincidence; it will always happen this way when the coefficients of α in the equations are even). So we have two candidates for the key: $(5, 15)$ and $(18, 15)$. However, $\gcd(18, 26) \neq 1$ so the second is ruled out. Therefore, the key is $(5, 15)$.

The preceding procedure works unless the gcd we get is 13 (or 26). In this case, use another letter of the message, if available.

If we know only one letter of plaintext, we still get a relation between α and β . For example, if we only know that g in plaintext corresponds to T in ciphertext, then we have $6\alpha + \beta \equiv 19 \pmod{26}$. There are 12 possibilities for α and each gives one corresponding β . Therefore, an exhaustive search through the 12 keys should yield the correct key.

3. Chosen plaintext: Choose ab as the plaintext. The first character of the ciphertext will be $\alpha \cdot 0 + \beta = \beta$, and the second will be $\alpha + \beta$. Therefore, we can find the key.
4. Chosen ciphertext: Choose AB as the ciphertext. This yields the decryption function of the form $x = \alpha_1 y + \beta_1$. We could solve for y and obtain the encryption key. But why bother? We have the decryption function, which is what we want.

2.3 The Vigenère Cipher

A variation of the shift cipher was invented back in the sixteenth century. It is often attributed to Vigenère, though Vigenère's encryption methods were more sophisticated. Well into the twentieth century, this cryptosystem was thought by many to be secure, though Babbage and Kasiski had shown how to attack it during the nineteenth century. In the 1920s, Friedman developed additional methods for breaking this and related ciphers.

The key for the encryption is a vector, chosen as follows. First choose a key length, for example, 6. Then choose a vector of this size whose entries are integers from 0 to 25, for example $k = (21, 4, 2, 19, 14, 17)$. Often the key corresponds to a word that is easily remembered. In our case, the word is *vector*. The security of the system depends on the fact that neither the keyword nor its length is known.

To encrypt the message using the k in our example, we take first the letter of the plaintext and shift by 21. Then shift the second letter by 4, the third by 2, and so on. Once we get to the end of the key, we start back at its first entry, so the seventh letter is shifted by 21, the eighth letter by 4, etc. Here is a diagram of the encryption process.

(plaintext)	h	e	r	e	i	s	h	o	w	i	t	w	o	r	k	s
(key)	21	4	2	19	14	17	21	4	2	19	14	17	21	4	2	19
(ciphertext)	C	I	T	X	W	J	C	S	Y	B	H	N	J	V	M	L

A known plaintext attack will succeed if enough characters are known since the key is simply obtained by subtracting the plaintext from the ciphertext mod 26. A chosen plaintext attack using the plaintext *aaaaaa* . . .

will yield the key immediately, while a chosen ciphertext attack with $AAAAA\dots$ yields the negative of the key. But suppose you have only the ciphertext. It was long thought that the method was secure against a ciphertext-only attack. However, it is easy to find the key in this case, too.

The cryptanalysis uses the fact that in most English texts the frequencies of letters are not equal. For example, e occurs much more frequently than x . These frequencies have been tabulated in [Beker-Piper] and are provided in [Table 2.1](#).

Table 2.1 Frequencies of Letters in English

a	b	c	d	e	f	g	h	i	j
.082	.015	.028	.043	.127	.022	.020	.061	.070	.002
k	l	m	n	o	p	q	r	s	t
.008	.040	.024	.067	.075	.019	.001	.060	.063	.091
u	v	w	x	y	z				
.028	.010	.023	.001	.020	.001				

[Table 2.1 Full Alternative Text](#)

Of course, variations can occur, though usually it takes a certain amount of effort to produce them. There is a book *Gadsby* by Ernest Vincent Wright that does not contain the letter e . Even more impressive is the book *La Disparition* by George Perec, written in French, which also does not have a single e (not only are there the usual problems with verbs, etc., but almost all feminine nouns and adjectives must be avoided). There is an English

translation by Gilbert Adair, A Void, which also does not contain *e*. But generally we can assume that the above gives a rough estimate of what usually happens, as long as we have several hundred characters of text.

If we had a simple shift cipher, then the letter *e*, for example, would always appear as a certain ciphertext letter, which would then have the same frequency as that of *e* in the original text. Therefore, a frequency analysis would probably reveal the key. However, in the preceding example of a Vigenère cipher, the letter *e* appears as both *I* and *X*. If we had used a longer plaintext, *e* would probably have been encrypted as each of *Z*, *I*, *G*, *X*, *S*, and *V*, corresponding to the shifts 21, 4, 2, 19, 14, 17. But the occurrences of *Z* in a ciphertext might not come only from *e*. The letter *v* is also encrypted to *Z* when its position in the text is such that it is shifted by 4. Similarly, *x*, *g*, *l*, and *i* can contribute *Z* to the ciphertext, so the frequency of *Z* is a combination of that of *e*, *v*, *x*, *g*, *l*, and *i* from the plaintext. Therefore, it appears to be much more difficult to deduce anything from a frequency count. In fact, the frequency counts are usually smoothed out and are much closer to 1/26 for each letter of ciphertext. At least, they should be much closer than the original distribution for English letters.

Here is a more substantial example. This example is also treated in Example 4 in the Computer Appendices. The ciphertext is the following:

VVHQWVVRHMUSGJGTHKIHTSSEJCHLSFCBGVWC
RLRYQTFSVGAHW
KCUHWAUGLQHNSLRLJSHLTSPISPRDXLJSVEEG
HLQWKASSKUWE
PWQTWVSPGOELKCQYFNSVWLJSNIQKGNGRGYBWL
WGOVIOKHKAZKQ
KXZGYHCECMIEIUJOQKWFWVEFQHKIJRCLRLKBIE
NQFRJLJSDHGR
HLSFQTWLAUQRHWDMWLGLUSGIKKFLRYVCWVSP

GPMLKASSJVOQXE
 GGVEYGGZMLJCXXLJSVPAIVWIKVRDRYGFRLJLSLV
 EGGVEYGGEI
 APUUISFPBTGNWWMUCZRVTWGLRWUGUMNCZVI
 LE

The frequencies are as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M
8	5	$\frac{1}{2}$	4	$\frac{1}{5}$	$\frac{1}{0}$	$\frac{2}{7}$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{7}$	$\frac{2}{5}$	7
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
7	5	9	$\frac{1}{4}$	$\frac{1}{7}$	$\frac{2}{4}$	8	$\frac{1}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	5	8	5

Note that there is no letter whose frequency is significantly larger than the others. As discussed previously, this is because *e*, for example, gets spread among several letters during the encryption process.

How do we decrypt the message? There are two steps: finding the key length and finding the key. In the following, we'll first show how to find the key length and then give one way to find the key. After an explanation of why the method for finding the key works, we give an alternative way to find the key.

2.3.1 Finding the Key Length

Write the ciphertext on a long strip of paper, and again on another long strip. Put one strip above the other, but displaced by a certain number of places (the potential key length). For example, for a displacement of two we have the following:

V	V	H	Q	W	V	V	R	H	M	U	S	G	J	G		
V	V	H	Q	W	V	V	R	H	M	U	S	G	J	G		
*																
T	H	K	I	H	T	S	S	E	J	C	H	L	S	F	C	B
K	I	H	T	S	S	E	J	C	H	L	S	F	C	B	G	V
*																
G	V	W	C	R	L	R	Y	Q	T	F	S	V	G	A	H	...
W	C	R	L	R	Y	Q	T	F	S	V	G	A	H	W	K	...
*																

Mark a * each time a letter and the one below it are the same, and count the total number of coincidences. In the text just listed, we have two coincidences so far. If we had continued for the entire ciphertext, we would have counted 14 of them. If we do this for different displacements, we obtain the following data:

displacement:	1	2	3	4	5	6
coincidences:	14	14	16	14	24	12

We have the most coincidences for a shift of 5. As we explain later, this is the best guess for the length of the key. This method works very quickly, even without a computer, and usually yields the key length.

2.3.2 Finding the Key: First Method

Now suppose we have determined the key length to be 5, as in our example. Look at the 1st, 6th, 11th, ... letters and

see which letter occurs most frequently. We obtain

A	B	C	D	E	F	G	H	I	J	K	L	M
o	o	7	1	1	2	9	o	1	8	8	o	o
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
3	o	4	5	2	o	3	6	5	1	o	1	o

The most frequent is G , though J , K , C are close behind. However, $J = e$ would mean a shift of 5, hence $C = x$. But this would yield an unusually high frequency for x in the plaintext. Similarly, $K = e$ would mean $P = j$ and $Q = k$, both of which have too high frequencies. Finally, $C = e$ would require $V = x$, which is unlikely to be the case. Therefore, we decide that $G = e$ and the first element of the key is $2 = c$.

We now look at the 2nd, 7th, 12th, ... letters. We find that G occurs 10 times and S occurs 12 times, and the other letters are far behind. If $G = e$, then $S = q$, which should not occur 12 times in the plaintext. Therefore, $S = e$ and the second element of the key is $14 = o$.

Now look at the 3rd, 8th, 13th, ... letters. The frequencies are

A	B	C	D	E	F	G	H	I	J	K	L	M
o	1	o	3	3	1	3	5	1	o	4	1 o	o
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
2	1	2	3	5	3	o	2	8	7	1	o	1

The initial guess that $L = e$ runs into problems; for example, $R = k$ and $E = x$ have too high frequency

and $A = t$ has too low. Similarly, $V = e$ and $W = e$ do not seem likely. The best choice is $H = e$ and therefore the third key element is $3 = d$.

The 4th, 9th, 14th, ... letters yield $4 = e$ as the fourth element of the key. Finally, the 5th, 10th, 15th, ... letters yield $18 = s$ as the final key element. Our guess for the key is therefore

$$\{2, 14, 3, 4, 18\} = \{c, o, d, e, s\}.$$

As we saw in the case of the 3rd, 8th, 13th, ... letters (this also happened in the 5th, 10th, 15th, ... case), if we take every fifth letter we have a much smaller sample of letters on which we are doing a frequency count. Another letter can overtake e in a short sample. But it is probable that most of the high-frequency letters appear with high frequencies, and most of the low-frequency ones appear with low frequencies. As in the present case, this is usually sufficient to identify the corresponding entry in the key.

Once a potential key is found, test it by using it to decrypt. It should be easy to tell whether it is correct.

In our example, the key is conjectured to be $(2, 14, 3, 4, 18)$. If we decrypt the ciphertext using this key, we obtain

themethodusedfortheprparationandreadingofcodemess
agesis

simpleintheextremeandalatthesametimeimpossibleoftransl
atio

nunlessthekeyisknowntheeasewithwhichthekeymaybech
angedis

anotherpointinfavoroftheadoptionofthiscodebythosedesi
rin

gtotransmitimportantmessagewithouttheslightestdange
roft

heirmessagesbeingreadbypoliticalorbusinessrivalsetc

This passage is taken from a short article in Scientific American, Supplement LXXXIII (January 27, 1917), page 61. A short explanation of the Vigenère cipher is given, and the preceding passage expresses an opinion as to its security.

Before proceeding to a second method for finding the key, we give an explanation of why the procedure given earlier finds the key length. In order to avoid confusion, we note that when we use the word “shift” for a letter, we are referring to what happens during the Vigenère encryption process.

We also will be shifting elements in vectors. However, when we slide one strip of paper to the right or left relative to the other strip, we use the word “displacement.”

Put the frequencies of English letters into a vector:

$$\mathbf{A}_0 = (.082, .015, .028, \dots, .020, .001).$$

Let \mathbf{A}_i be the result of shifting \mathbf{A}_0 by i spaces to the right. For example,

$$\mathbf{A}_2 = (.020, .001, .082, .015, \dots).$$

The dot product of \mathbf{A}_0 with itself is

$$\mathbf{A}_0 \cdot \mathbf{A}_0 = (.082)^2 + (.015)^2 + \dots = .066.$$

Of course, $\mathbf{A}_i \cdot \mathbf{A}_i$ is also equal to .066 since we get the same sum of products, starting with a different term. However, the dot products of $\mathbf{A}_i \cdot \mathbf{A}_j$ are much lower when $i \neq j$, ranging from .031 to .045:



$ i - j $	0	1	2	3	4	5	6
$\mathbf{A}_i \cdot \mathbf{A}_j$.06 6	.03 9	.03 2	.03 4	.04 4	.03 3	.03 6
	7	8	9	10	11	12	13
	.03 9	.03 4	.03 4	.03 8	.04 5	.03 9	.04 2

The dot product depends only on $|i - j|$. This can be seen as follows. The entries in the vectors are the same as those in \mathbf{A}_0 , but shifted. In the dot product, the i th entry of \mathbf{A}_0 is multiplied by the j th entry, the $(i + 1)$ st times the $(j + 1)$ st, etc. So each element is multiplied by the element $j - i$ positions removed from it. Therefore, the dot product depends only on the difference $i - j$.

However, by reversing the roles of i and j , and noting that $\mathbf{A}_i \cdot \mathbf{A}_j = \mathbf{A}_j \cdot \mathbf{A}_i$, we see that $i - j$ and $j - i$ give the same dot products, so the dot product only depends on $|i - j|$. In the preceding table, we only needed to compute up to $|i - j| = 13$. For example, $i - j = 17$ corresponds to a shift by 17 in one direction, or 9 in the other direction, so $i - j = 9$ will give the same dot product.

The reason $\mathbf{A}_0 \cdot \mathbf{A}_0$ is higher than the other dot products is that the large numbers in the vectors are paired with large numbers and the small ones are paired with small. In the other dot products, the large numbers are paired somewhat randomly with other numbers. This lessens their effect. For another reason that $\mathbf{A}_0 \cdot \mathbf{A}_0$ is higher than the other dot products, see [Exercise 23](#).

Let's assume that the distribution of letters in the plaintext closely matches that of English, as expressed by the vector \mathbf{A}_0 above. Look at a random letter in the top strip of ciphertext. It corresponds to a random letter of English shifted by some amount i (corresponding to an element of the key). The letter below it corresponds to a random letter of English shifted by some amount j .

For concreteness, let's suppose that $i = 0$ and $j = 2$. The probability that the letter in the 50th position, for example, is A is given by the first entry in \mathbf{A}_0 , namely .082. The letter directly below, on the second strip, has been shifted from the original plaintext by $j = 2$ positions. If this ciphertext letter is A , then the corresponding plaintext letter was y , which occurs in the plaintext with probability .020. Note that .020 is the first entry of the vector \mathbf{A}_2 . The probability that the letter in the 50th position on the first strip and the letter directly below it are both the letter A is $(.082)(.020)$. Similarly, the probability that both letters are B is $(.015)(.001)$. Working all the way through Z , we see that the probability that the two letters are the same is

$$(.082)(.020) + (.015)(.001) + \dots + (.001)(.001) = \mathbf{A}_0 \cdot \mathbf{A}_2.$$

In general, when the encryption shifts are i and j , the probability that the two letters are the same is $\mathbf{A}_i \cdot \mathbf{A}_j$. When $i \neq j$, this is approximately 0.038, but if $i = j$, then the dot product is 0.066.

We are in the situation where $i = j$ exactly when the letters lying one above the other have been shifted by the same amount during the encryption process, namely when the top strip is displaced by an amount equal to the key length (or a multiple of the key length). Therefore we expect more coincidences in this case.

For a displacement of 5 in the preceding ciphertext, we had 326 comparisons and 24 coincidences. By the reasoning just given, we should expect approximately $326 \times 0.066 = 21.5$ coincidences, which is close to the actual value.

2.3.3 Finding the Key: Second Method

Using the preceding ideas, we give another method for determining the key. It seems to work somewhat better than the first method on short samples, though it requires a little more calculation.

We'll continue to work with the preceding example. To find the first element of the key, count the occurrences of the letters in the 1st, 6th, 11th, ... positions, as before, and put them in a vector:

$$\mathbf{V} = (0, 0, 7, 1, 1, 2, 9, 0, 1, 8, 8, 0, 0, 3, 0, 4, 5, 2, 0, 3, 6, 5, 1, 0, 1, 0)$$

(the first entry gives the number of occurrences of A , the second gives the number of occurrences of B , etc.). If we divide by 67, which is the total number of letters counted, we obtain a vector

$$\mathbf{W} = (0, 0, .1045, .0149, .0149, .0299, \dots, .0149, 0).$$

Let's think about where this vector comes from. Since we know the key length is 5, the 1st, 6th, 11th, ... letters in the ciphertext were all shifted by the same amount (as we'll see shortly, they were all shifted by 2). Therefore, they represent a random sample of English letters, all shifted by the same amount. Their frequencies, which are given by the vector \mathbf{W} , should approximate the vector \mathbf{A}_i , where i is the shift caused by the first element of the key.

The problem now is to determine i . Recall that $\mathbf{A}_i \cdot \mathbf{A}_j$ is largest when $i = j$, and that \mathbf{W} approximates \mathbf{A}_i . If we compute $\mathbf{W} \cdot \mathbf{A}_j$ for $0 \leq j \leq 25$, the maximum value should occur when $j = i$. Here are the dot products:

$$\begin{aligned} &.0250, .0391, .0713, .0388, .0275, .0380, .0512, .0301, .0325, \\ &.0430, .0338, .0299, .0343, .0446, .0356, .0402, .0434, .0502, \\ &.0392, .0296, .0326, .0392, .0366, .0316, .0488, .0349 \end{aligned}$$

The largest value is the third, namely $.0713$, which equals $\mathbf{W} \cdot \mathbf{A}_2$. Therefore, we guess that the first shift is 2, which corresponds to the key letter c .

Let's use the same method to find the third element of the key. We calculate a new vector \mathbf{W} , using the frequencies for the 3rd, 8th, 13th, ... letters that we tabulated previously:

$$\mathbf{W} = (0, .0152, 0, .0454, .0454, .0152, \dots, 0, .0152).$$

The dot products $\mathbf{W} \cdot \mathbf{A}_i$ for $0 \leq i \leq 25$ are

$$\begin{aligned} &.0372, .0267, .0395, .0624, .04741, .0279, .0319, .0504, .0378, \\ &.0351, .0367, .0395, .0264, .0415, .0427, .0362, .0322, .0457, \\ &.0526, .0397, .0322, .0299, .0364, .0372, .0352, .0406 \end{aligned}$$

The largest of these values is the fourth, namely $.0624$, which equals $\mathbf{W} \cdot \mathbf{A}_3$. Therefore, the best guess is that the first shift is 3, which corresponds to the key letter d . The other three elements of the key can be found similarly, again yielding c, o, d, e, s as the key.

Notice that the largest dot product was significantly larger than the others in both cases, so we didn't have to make several guesses to find the correct one. In this way, the present method is superior to the first method presented; however, the first method is much easier to do by hand.

Why is the present method more accurate than the first one? To obtain the largest dot product, several of the larger values in \mathbf{W} had to match with the larger values in an \mathbf{A}_i . In the earlier method, we tried to match only the e , then looked at whether the choices for other letters were reasonable. The present method does this all in one step.

To summarize, here is the method for finding the key. Assume we already have determined that the key length is n .

For $i = 1$ to n , do the following:

1. Compute the frequencies of the letters in positions $i \bmod n$, and form the vector \mathbf{W} .

2. For $j = 0$ to 25 , compute $\mathbf{W} \cdot \mathbf{A}_j$.

3. Let $k_i = j_0$ give the maximum value of $\mathbf{W} \cdot \mathbf{A}_j$.

The key is probably $\{k_1, \dots, k_n\}$.

2.4 Substitution Ciphers

One of the more popular cryptosystems is the substitution cipher. It is commonly used in the puzzle section of the weekend newspapers, for example. The principle is simple: Each letter in the alphabet is replaced by another (or possibly the same) letter. More precisely, a permutation of the alphabet is chosen and applied to the plaintext. In the puzzle pages, the spaces between the words are usually preserved, which is a big advantage to the solver, since knowledge of word structure becomes very useful. However, to increase security it is better to omit the spaces.

The shift and affine ciphers are examples of substitution ciphers. The Vigenère cipher (see [Section 2.3](#)) is not, since it permutes blocks of letters rather than one letter at a time.

Everyone “knows” that substitution ciphers can be broken by frequency counts. However, the process is more complicated than one might expect.

Consider the following example. Thomas Jefferson has a potentially treasonous message that he wants to send to Ben Franklin. Clearly he does not want the British to read the text if they intercept it, so he encrypts it using a substitution cipher. Fortunately, Ben Franklin knows the permutation being used, so he can simply reverse the permutation to obtain the original message (of course, Franklin was quite clever, so perhaps he could have decrypted it without previously knowing the key).

Now suppose we are working for the Government Code and Cypher School in England back in 1776 and are given the following intercepted message to decrypt.

LWNSOZBNWVBAYBNVBSQWVWOHWDIZWRBBN
PBPOOUWRPAWXAW

PBWZWMYPOBNPBBNWJPAWWRZSLWZQJBNIAX
AWPBSALIBNXWA

BPIRYRPOIWRPQOWAIENVBVNPBPUSREBNWVWP
WOIHWOIQWAB

JPRZBNWFYAVYIBSHNPFFIRWVVBNPBBSVWXYAW
BNWVVAIENBV

ESDWARUWRBVPAWIRVBIBYWZPUSREUWRZWAI
DIREBNWIATYY

BFSLWAVHASUBNWXSRVWRBSHBNWESDWARWZB
NPBLNWRWDWAPR

JHS AUSH ESD WARU WRBQWX SUWVZ WBAY X BIDW
SHBNW VVW RZ VIB

IVBNWAIENBSHBNWFWSFOWBSPOBWASABSPQSOI
VNIBPRZBSIR

VBIBYBWRWLESDWARUWRBOPJIREIBVHSYRZPBIS
RSRVYXNFAI

RXIFOWVPRZSAEPRIKIREIBVFSLWAVIRVYXNHSAU
PVBSVWWUU

SVBOICWOJBSWHWXBBNWIAPPHWBPRZNPFFI
RWVV

A frequency count yields the following (there are 520 letters in the text):

W	B	R	S	I	V	A	P	N	O	...
76	6 4	3 9	3 6	3 6	3 5	3 4	3 2	3 0	1 6	...

The approximate frequencies of letters in English were given in [Section 2.3](#). We repeat some of the data here in [Table 2.2](#). This allows us to guess with reasonable confidence that W represents e (though B is another possibility). But what about the other letters? We can guess that B, R, S, I, V, A, P, N , with maybe an exception or two, are probably the same as t, a, o, i, n, s, h, r in some order. But a simple frequency count is not enough to decide which is which. What we need to do now is look at digrams, or pairs of letters. We organize our results in [Table 2.3](#) (we only use the most frequent letters here, though it would be better to include all).

Table 2.2 Frequencies of Most Common Letters in English

e	t	a	o	i	n	s	h	r
.127	.091	.082	.075	.070	.067	.063	.061	.060

[Table 2.2 Full Alternative Text](#)

Table 2.3 Counting Digrams

	W	B	R	S	I	V	A	P	N
W	3	4	12	2	4	10	14	3	1
B	4	4	0	11	5	5	2	4	20
R	5	5	0	1	1	5	0	3	0
S	1	0	5	0	1	3	5	2	0
I	1	8	10	1	0	2	3	0	0
V	8	10	0	0	2	2	0	3	1
A	7	3	4	2	5	4	0	1	0
P	0	8	6	0	1	1	4	0	0
N	14	3	0	1	1	1	0	7	0

Table 2.3 Full Alternative Text

The entry 1 in the W row and N column means that the combination WN appears 1 time in the text. The entry 14 in the N row and W column means that NW appears 14 times.

We have already decided that $W = e$, but if we had extended the table to include low-frequency letters, we would see that W contacts many of these letters, too, which is another characteristic of e . This helps to confirm our guess.

The vowels a, i, o tend to avoid each other. If we look at the R row, we see that R does not precede S, I, A, N very often. But a look at the R column shows that R follows S, I, A fairly often. So we suspect that R is not one of a, i, o . V and N are out because they would require a, i , or o to precede $W = e$ quite often, which is unlikely. Continuing, we see that the most likely possibilities for a, i, o are S, I, P in some order.

The letter n has the property that around 80% of the letters that precede it are vowels. Since we already have identified W, S, I, P as vowels, we see that R and A are the most likely candidates. We'll have to wait to see which is correct.

The letter h often appears before e and rarely after it.
This tells us that $N = h$.

The most common digram is th . Therefore, $B = t$.

Among the frequent letters, r and s remain, and they should equal V and one of A , R . Since r pairs more with vowels and s pairs more with consonants, we see that V must be s and r is represented by either A or R .

The combination rn should appear more than nr , and AR is more frequent than RA , so our guess is that $A = r$ and $R = n$.

We can continue the analysis and determine that $S = o$ (note that to is much more common than ot), $I = i$, and $P = a$ are the most likely choices. We have therefore determined reasonable guesses for 382 of the 520 characters in the text:

L	W	N	S	O	Z	B	N	W	V	W	B	A	Y	B	N	V	B	S
e	h	o				t	h	e	s	e	t	r		t	h	s	t	o
Q	W	V	W	O	H	W	D	I	Z	W	R	B	B	N	P	B	P	...
e	s	e				e		i		e	n	t	t	h	a	t	a	...

At this point, knowledge of the language, middle-level frequencies (l , d , \dots), and educated guesses can be used to fill in the remaining letters. For example, in the first line a good guess is that $Y = u$ since then the word *truths* appears. Of course, there is a lot of guesswork, and various hypotheses need to be tested until one works.

Since the preceding should give the spirit of the method, we skip the remaining details. The decrypted message, with spaces (but not punctuation) added, is as follows

(the text is from the middle of the Declaration of Independence):

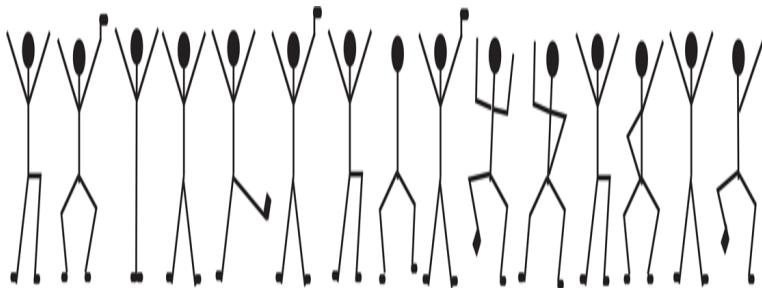
we hold these truths to be self evident that all men are created equal that they are endowed by their creator with certain unalienable rights that among these are life liberty and the pursuit of happiness that to secure these rights governments are instituted among men deriving their just powers from the consent of the governed that whenever any form of government becomes destructive of these ends it is the right of the people to alter or to abolish it and to institute new government laying its foundation on such principles and organizing its powers in such form as to seem most likely to effect their safety and happiness

2.5 Sherlock Holmes

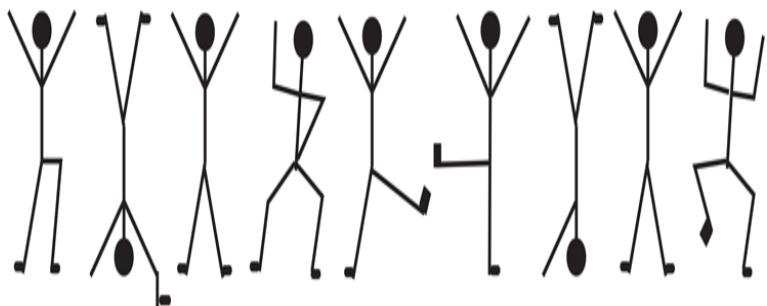
Cryptography has appeared in many places in literature, for example, in the works of Edgar Allan Poe (*The Gold Bug*), William Thackeray (*The History of Henry Esmond*), Jules Verne (*Voyage to the Center of the Earth*), and Agatha Christie (*The Four Suspects*).

Here we give a summary of an enjoyable tale by Arthur Conan Doyle, in which Sherlock Holmes displays his usual cleverness, this time by breaking a cipher system. We cannot do the story justice here, so we urge the reader to read *The Adventure of the Dancing Men* in its entirety. The following is a cryptic, and cryptographic, summary of the plot.

Mr. Hilton Cubitt, who has recently married the former Elsie Patrick, mails Sherlock Holmes a letter. In it is a piece of paper with dancing stick figures that he found in his garden at Riding Thorpe Manor:



Two weeks later, Cubitt finds another series of figures written in chalk on his toolhouse door:



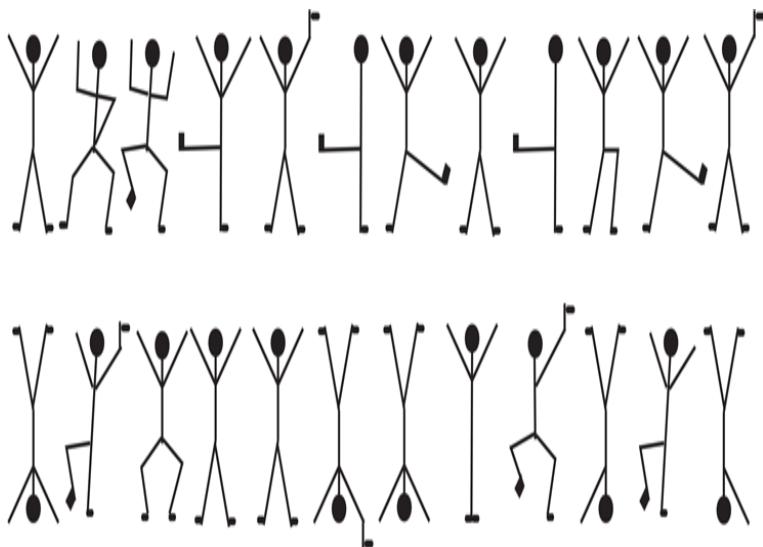
Two mornings later another sequence appears:



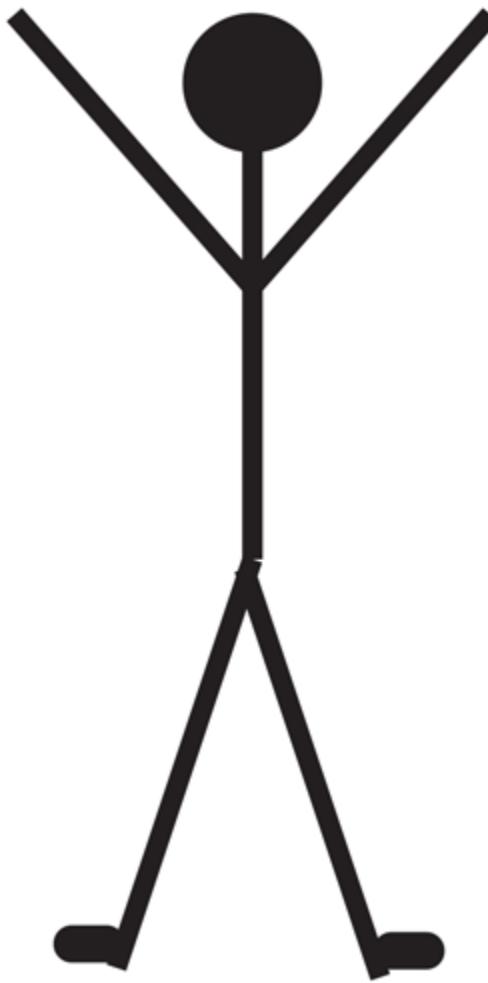
Three days later, another message appears:



Cubitt gives copies of all of these to Holmes, who spends the next two days making many calculations. Suddenly, Holmes jumps from his chair, clearly having made a breakthrough. He quickly sends a long telegram to someone and then waits, telling Watson that they will probably be going to visit Cubitt the next day. But two days pass with no reply to the telegram, and then a letter arrives from Cubitt with yet another message:



Holmes studies it and says they need to travel to Riding Thorpe Manor as soon as possible. A short time later, a reply to Holmes's telegram arrives, and Holmes indicates that the matter has become even more urgent. When Holmes and Watson arrive at Cubitt's house the next day, they find the police already there. Cubitt has been shot dead. His wife, Elsie, has also been shot and is in critical condition (although she survives). Holmes asks several questions and then has someone deliver a note to a Mr. Abe Slaney at nearby Elrige's Farm. Holmes then explains to Watson and the police how he decrypted the messages. First, he guessed that the flags on some of the figures indicated the ends of words. He then noticed that the most common figure was



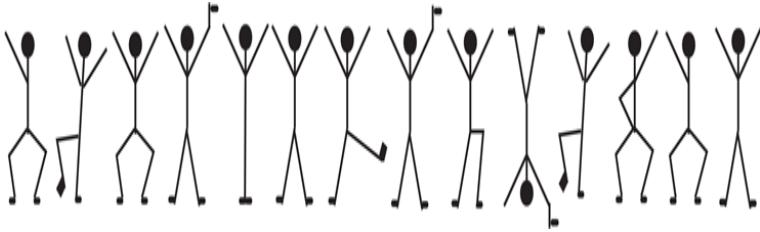
so it was likely *E*. This gave the fourth message as *-E-E-*. The possibilities *LEVER*, *NEVER*, *SEVER* came to mind, but since the message was probably a one word reply to a previous message, Holmes guessed it was *NEVER*. Next, Holmes observed that



had the form *E---E*, which could be *ELSIE*. The third message was therefore *--E ELSIE*. Holmes tried

several combinations, finally settling on *COME ELSIE* as the only viable possibility. The first message therefore was – *M –ERE – –E SL–NE–*. Holmes guessed that the first letter was *A* and the third letter as *H*, which gave the message as *AM HERE A–E SLANE–*. It was reasonable to complete this to *AM HERE ABE SLANEY*. The second message then was *A– ELRI–ES*. Of course, Holmes correctly guessed that this must be stating where Slaney was staying. The only letters that seemed reasonable completed the phrase to *AT ELRIGES*. It was after decrypting these two messages that Holmes sent a telegram to a friend at the New York Police Bureau, who sent back the reply that Abe Slaney was “the most dangerous crook in Chicago.” When the final message arrived, Holmes decrypted it to *ELSIE –RE–ARE TO MEET THY GO–*. Since he recognized the missing letters as *P, P, D*, respectively, Holmes became very concerned and that’s why he decided to make the trip to Riding Thorpe Manor.

When Holmes finishes this explanation, the police urge that they go to Elrige’s and arrest Slaney immediately. However, Holmes suggests that is unnecessary and that Slaney will arrive shortly. Sure enough, Slaney soon appears and is handcuffed by the police. While waiting to be taken away, he confesses to the shooting (it was somewhat in self-defense, he claims) and says that the writing was invented by Elsie Patrick’s father for use by his gang, the Joint, in Chicago. Slaney was engaged to be married to Elsie, but she escaped from the world of gangsters and fled to London. Slaney finally traced her location and sent the secret messages. But why did Slaney walk into the trap that Holmes set? Holmes shows the message he wrote:



From the letters already deduced, we see that this says *COME HERE AT ONCE*. Slaney was sure this message must have been from Elsie since he was certain no one outside of the Joint could write such messages. Therefore, he made the visit that led to his capture.

Comments

What Holmes did was solve a simple substitution cipher, though he did this with very little data. As with most such ciphers, both frequency analysis and a knowledge of the language are very useful. A little luck is nice, too, both in the form of lucky guesses and in the distribution of letters. Note how overwhelmingly *E* was the most common letter. In fact, it appeared 11 times among the 38 characters in the first four messages. This gave Holmes a good start. If Elsie had been Carol and Abe Slaney had been John Smith, the decryption would probably have been more difficult.

Authentication is an important issue in cryptography. If Eve breaks Alice's cryptosystem, then Eve can often masquerade as Alice in communications with Bob. Safeguards against this are important. The judges gave Abe Slaney many years to think about this issue.

The alert reader might have noticed that we cheated a little when decrypting the messages. The same symbol represents the *V* in *NEVER* and the *Ps* in *PREPARE*. This is presumably due to a misprint and has occurred in every printed version of the work, starting with the story's first publication back in 1903. In the original text,

the *R* in *NEVER* is written as the *B* in *ABE*, but this is corrected in later editions (however, in some later editions, the first *C* in the message Holmes wrote is given an extra arm and therefore looks like the *M*). If these mistakes had been in the text that Holmes was working with, he would have had a very difficult time decrypting and would have rightly concluded that the Joint needed to use error correction techniques in their transmissions. In fact, some type of error correction should be used in conjunction with almost every cryptographic protocol.

2.6 The Playfair and ADFGX Ciphers

The Playfair and ADFGX ciphers were used in World War I by the British and the Germans, respectively. By modern standards, they are fairly weak systems, but they took real effort to break at the time.

The Playfair system was invented around 1854 by Sir Charles Wheatstone, who named it after his friend, the Baron Playfair of St. Andrews, who worked to convince the government to use it. In addition to being used in World War I, it was used by the British forces in the Boer War.

The key is a word, for example, *playfair*. The repeated letters are removed, to obtain *playfir*, and the remaining letters are used to start a 5×5 matrix. The remaining spaces in the matrix are filled in with the remaining letters in the alphabet, with *i* and *j* being treated as one letter:

<i>p</i>	<i>l</i>	<i>a</i>	<i>y</i>	<i>f</i>
<i>i</i>	<i>r</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>e</i>	<i>g</i>	<i>h</i>	<i>k</i>	<i>m</i>
<i>n</i>	<i>o</i>	<i>q</i>	<i>s</i>	<i>t</i>
<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>z</i>

Suppose the plaintext is *meet at the schoolhouse*. Remove spaces and divide the text into groups of two letters. If there is a doubled letter appearing as a group, insert an *x* and regroup. Add an extra *x* at the end to complete the last group, if necessary. Our plaintext becomes

me et at th es ch ox ol ho us ex.

Now use the matrix to encrypt each two-letter group by the following scheme:

- If the two letters are not in the same row or column, replace each letter by the letter that is in its row and is in the column of the other letter. For example, *et* becomes *MN*, since *M* is in the same row as *e* and the same column as *t*, and *N* is in the same row as *t* and the same column as *e*.
- If the two letters are in the same row, replace each letter with the letter immediately to its right, with the matrix wrapping around from the last column to the first. For example, *me* becomes *EG*.
- If the two letters are in the same column, replace each letter with the letter immediately below it, with the matrix wrapping around from the last row to the first. For example, *ol* becomes *VR*.

The ciphertext in our example is

EG MN FQ QM KN BK SV VR GQ XN KU.

To decrypt, reverse the procedure.

The system succumbs to a frequency attack since the frequencies of the various digrams (two-letter combinations) in English have been tabulated. Of course, we only have to look for the most common digrams; they should correspond to the most common digrams in English: *th*, *he*, *an*, *in*, *re*, *es*, Moreover, a slight modification yields results more quickly. For example, both of the digrams *re* and *er* are very common. If the pairs *IG* and *GI* are common in the ciphertext, then a good guess is that *e*, *i*, *r*, *g* form the corners of a rectangle in the matrix. Another weakness is that each plaintext letter has only five possible corresponding ciphertext letters. Also, unless the keyword is long, the last few rows of the matrix are predictable. Observations such as these allow the system to be broken with a ciphertext-only attack. For more on its cryptanalysis, see [Gaines].

The ADFGX cipher proceeds as follows. Put the letters of the alphabet into a 5×5 matrix. The letters *i* and *j* are treated as one, and the columns of the matrix are labeled

with the letters A, D, F, G, X . For example, the matrix could be

	A	D	F	G	X
A	p	g	c	e	n
D	b	q	o	z	r
F	s	l	a	f	t
G	m	d	v	i	w
X	k	u	y	x	h

2.6-12 Full Alternative Text

Each plaintext letter is replaced by the label of its row and column. For example, s becomes FA , and z becomes DG . Suppose the plaintext is

Kaiser Wilhelm.

The result of this initial step is

$X A F F G G F A A G D X G X G G F D X X A G F D G A.$

So far, this is a disguised substitution cipher. The next step increases the complexity significantly. Choose a keyword, for example, *Rhein*. Label the columns of a matrix by the letters of the keyword and put the result of the initial step into another matrix:

R	H	E	I	N
X	A	F	F	G
G	F	A	A	G
D	X	G	X	G
G	F	D	X	X
A	G	F	D	G
A				

2.6-13 Full Alternative Text

Now reorder the columns so that the column labels are in alphabetic order:

<i>E</i>	<i>H</i>	<i>I</i>	<i>N</i>	<i>R</i>
<i>F</i>	<i>A</i>	<i>F</i>	<i>G</i>	<i>X</i>
<i>A</i>	<i>F</i>	<i>A</i>	<i>G</i>	<i>G</i>
<i>G</i>	<i>X</i>	<i>X</i>	<i>G</i>	<i>D</i>
<i>D</i>	<i>F</i>	<i>X</i>	<i>X</i>	<i>G</i>
<i>F</i>	<i>G</i>	<i>D</i>	<i>G</i>	<i>A</i>
				<i>A</i>

2.6-14 Full Alternative Text

Finally, the ciphertext is obtained by reading down the columns (omitting the labels) in order:

FAGDFAFXFGFAXXDGGGXGXGDGAA.

Decryption is easy, as long as you know the keyword. From the length of the keyword and the length of the ciphertext, the length of each column is determined. The letters are placed into columns, which are reordered to match the keyword. The original matrix is then used to recover the plaintext.

The initial matrix and the keyword were changed frequently, making cryptanalysis more difficult, since there was only a limited amount of ciphertext available for any combination. However, the system was successfully attacked by the French cryptanalyst Georges Painvin and the Bureau du Chiffre, who were able to decrypt a substantial number of messages.

Here is one technique that was used. Suppose two different ciphertexts intercepted at approximately the same time agree for the first several characters. A reasonable guess is that the two plaintexts agree for several words. That means that the top few entries of the

columns for one are the same as for the other. Search through the ciphertexts and find other places where they agree. These possibly represent the beginnings of the columns. If this is correct, we know the column lengths. Divide the ciphertexts into columns using these lengths. For the first ciphertext, some columns will have one length and others will be one longer. The longer ones represent columns that should be near the beginning; the other columns should be near the end. Repeat for the second ciphertext. If a column is long for both ciphertexts, it is very near the beginning. If it is long for one ciphertext and not for the other, it goes in the middle. If it is short for both, it is near the end. At this point, try the various orderings of the columns, subject to these restrictions. Each ordering corresponds to a potential substitution cipher. Use frequency analysis to try to solve these. One should yield the plaintext, and the initial encryption matrix.

The letters *ADFGX* were chosen because their symbols in Morse code ($\cdot\cdot$, $- \cdot \cdot$, $\cdot \cdot - \cdot$, $- - \cdot$, $- \cdot \cdot -$) were not easily confused. This was to avoid transmission errors, and represents one of the early attempts to combine error correction with cryptography. Eventually, the *ADFGX* cipher was replaced by the *ADFGVX* cipher, which used a 6×6 initial matrix. This allowed all 26 letters plus 10 digits to be used.

For more on the cryptanalysis of the ADFGX cipher, see [Kahn].

2.7 Enigma

Mechanical encryption devices known as rotor machines were developed in the 1920s by several people. The best known was designed by Arthur Scherbius and became the famous Enigma machine used by the Germans in World War II.

It was believed to be very secure and several attempts at breaking the system ended in failure. However, a group of three Polish cryptologists, Marian Rejewski, Henryk Zygalski, and Jerzy Różycki, succeeded in breaking early versions of Enigma during the 1930s. Their techniques were passed to the British in 1939, two months before Germany invaded Poland. The British extended the Polish techniques and successfully decrypted German messages throughout World War II.

The fact that Enigma had been broken remained a secret for almost 30 years after the end of the war, partly because the British had sold captured Enigma machines to former colonies and didn't want them to know that the system had been broken.

In the following, we give a brief description of Enigma and then describe an attack developed by Rejewski. For more details, see for example [Kozaczuk], which contains appendices by Rejeweski giving details of attacks on Enigma.

We give a basic schematic diagram of the machine in [Figure 2.1](#). For more details, we urge the reader to visit some of the many websites that can be found on the Internet that give pictures of actual Enigma machines and extensive diagrams of the internal workings of these machines. There are also several online Enigma

simulators. Try one of them to get a better understanding of how Enigma works.

Figure 2.1 A Schematic Diagram of the Enigma Machine

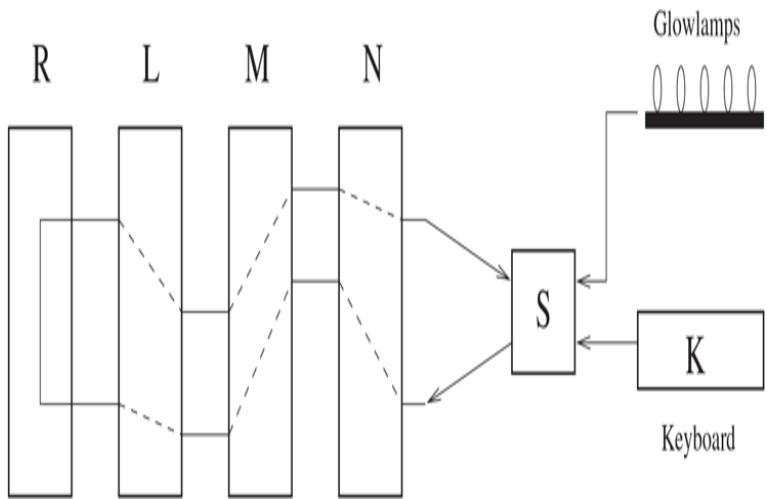


Figure 2.1 Full Alternative Text

L, M, N are the rotors. On one side of each rotor are 26 fixed electrical contacts, arranged in a circle. On the other side are 26 spring-loaded contacts, again arranged in a circle so as to touch the fixed contacts of the adjacent rotor. Inside each rotor, the fixed contacts are connected to the spring-loaded contacts in a somewhat random manner. These connections are different in each rotor. Each rotor has 26 possible initial settings.

R is the reversing drum. It has 26 spring-loaded contacts, connected in pairs.

K is the keyboard and is the same as a typewriter keyboard.

S is the plugboard. It has approximately six pairs of plugs that can be used to interchange six pairs of letters.

When a key is pressed, the first rotor N turns $1/26$ of a turn. Then, starting from the key, electricity passes through S , then through the rotors N, M, L . When it reaches the reversing drum R , it is sent back along a different path through L, M, N , then through S . At this point, the electricity lights a bulb corresponding to a letter on the keyboard, which is the letter of the ciphertext.

Since the rotor N rotates before each encryption, this is much more complicated than a substitution cipher. Moreover, the rotors L and M also rotate, but much less often, just like the wheels on a mechanical odometer.

Decryption uses exactly the same method. Suppose a sender and receiver have identical machines, both set to the same initial positions. The sender encrypts the message by typing it on the keyboard and recording the sequence of letters indicated by the lamps. This ciphertext is then sent to the receiver, who types the ciphertext into the machine. The sequence of letters appearing in the lamps is the original message. This can be seen as follows. Lamp “a” and key “a” are attached to a wire coming out of the plugboard. Lamp “h” and key “h” are attached to another wire coming out of the plugboard. If the key “a” is pressed and the lamp “h” lights up, then the electrical path through the machine is also connecting lamp “a” to key “h”. Therefore, if the “h” key were pressed instead, then the “a” key would light.

Similar reasoning shows that no letter is ever encrypted as itself. This might appear to be a good idea, but actually it is a weakness since it allows a cryptanalyst to discard many possibilities at the start. See [Chapter 14](#).

The security of the system rests on the keeping secret the initial settings of the rotors, the setting of the plugs on the plugboard, and the internal wiring of the rotors and

reversing drum. The settings of the rotors and the plugboard are changed periodically (for example, daily).

We'll assume the internal wiring of the rotors is known. This would be the case if a machine were captured, for example. However, there are ways to deduce this information, given enough ciphertext, and this is what was actually done in some cases.

How many combinations of settings are there? There are 26 initial settings for each of the three rotors. This gives $26^3 = 17576$ possibilities. There are six possible orderings of the three rotors. This yields $6 \times 17576 = 105456$ possible ways to initialize the rotors. In later versions of Enigma, there were five rotors available, and each day three were chosen. This made 60 possible orderings of the rotors and therefore 1054560 ways to initialize the rotors.

On the plugboard, there are 100391791500 ways of interchanging six pairs of letters.

In all, there seem to be too many possible initializations of the machine to have any hope of breaking the system. Techniques such as frequency analysis fail since the rotations of the rotors change the substitution for each character of the message.

So, how was Enigma attacked? We don't give the whole attack here, but rather show how the initial settings of the rotors were determined in the years around 1937. This attack depended on a weakness in the protocol being used at that time, but it gives the general flavor of how the attacks proceeded in other situations.

Each Enigma operator was given a codebook containing the daily settings to be used for the next month. However, if these settings had been used without modification, then each message sent during a given day would have had its first letter encrypted by the same

substitution cipher. The rotor would then have turned and the second letter of each text would have corresponded to another substitution cipher, and this substitution would have been the same for all messages for that day. A frequency analysis on the first letter of each intercepted message during a day would probably allow a decryption of the first letter of each text. A second frequency analysis would decrypt the second letters. Similarly, the remaining letters of the ciphertexts (except for the ends of the longest few ciphertexts) could be decrypted.

To avoid this problem, for each message the operator chose a message key consisting of a sequence of three letters, for example, *r*, *f*, *u*. He then used the daily setting from the codebook to encrypt this message key. But since radio communications were prone to error, he typed in *rfu* twice, therefore encrypting *rfurfu* to obtain a string of six letters. The rotors were then set to positions *r*, *f*, and *u* and the encryption of the actual message began. So the first six letters of the transmitted message were the encrypted message key, and the remainder was the ciphertext. Since each message used a different key, frequency analysis didn't work.

The receiver simply used the daily settings from the codebook to decrypt the first six letters of the message. He then reset the rotors to the positions indicated by the decrypted message key and proceeded to decrypt the message.

The duplication of the key was a great aid to the cryptanalysts. Suppose that on some day you intercept several messages, and among them are three that have the following initial six letters:

dmqvbn

vonpuy

pucfmq

All of these were encrypted with the same daily settings from the codebook. The first encryption corresponds to a permutation of the 26 letters; let's call this permutation A . Before the second letter is encrypted, a rotor turns, so the second letter uses another permutation; call it B . Similarly, there are permutations C, D, E, F for the remaining four letters. The strategy is to look at the products AD, BE , and CF .

We need a few conventions and facts about permutations. When we write AD for two permutations A and D , we mean that we apply the permutation A then D (some books use the reverse ordering). The permutation that maps a to b , b to c , and c to a will be denoted as the 3-cycle (abc) . A similar notation will be used for cycles of other lengths. For example, (ab) is the permutation that switches a and b . A permutation can be written as a product of cycles. For example, the permutation

$$(dvpfkxgzyo)(eijmunqlht)(bc)(rw)(a)(s)$$

is the permutation that maps d to v , v to p , t to e , r to w , etc., and fixes a and s . If the cycles are disjoint (meaning that no two cycles have letters in common), then this decomposition into cycles is unique.

Let's look back at the intercepted texts. We don't know the letters of any of the three message keys, but let's call the first message key xyz . Therefore, $xyzzxyz$ encrypts to $dmqvbn$. We know that permutation A sends x to d . Also, the fourth permutation D sends x to v . But we know more. Because of the internal wiring of the machine, A actually interchanges x and d and D interchanges x and v . Therefore, the product of the permutations, AD , sends d to v (namely, A sends d to x and then D sends x to v). The unknown x has been eliminated. Similarly, the second intercepted text tells us

that AD sends v to p , and the third tells us that AD sends p to f . We have therefore determined that

$$AD = (dvpf \cdots) \cdots$$

In the same way, the second and fifth letters of the three messages tell us that

$$BE = (oumb \cdots) \cdots$$

and the third and sixth letters tell us that

$$CF = (cqny \cdots) \cdots$$

With enough data, we can deduce the decompositions of AD , BE , and CF into products of cycles. For example, we might have

$$\begin{aligned} AD &= (dvpfkxgzyo)(eijmunqlht)(bc)(rw)(a)(s) \\ BE &= (blfqveoum)(hjpswizrn)(axt)(cgy)(d)(k) \\ CF &= (abviktjgfqny)(duzrehlxwpsmo). \end{aligned}$$

This information depends only on the daily settings of the plugboard and the rotors, not on the message key. Therefore, it relates to every machine used on a given day.

Let's look at the effect of the plugboard. It introduces a permutation S at the beginning of the process and then adds the inverse permutation S^{-1} at the end. We need another fact about permutations: Suppose we take a permutation P and another permutation of the form SPS^{-1} for some permutation S (where S^{-1} denotes the inverse permutation of S ; in our case, $S = S^{-1}$) and decompose each into cycles. They will usually not have the same cycles, but the lengths of the cycles in the decompositions will be the same. For example, AD has cycles of length 10, 10, 2, 2, 1, 1. If we decompose $SADS^{-1}$ into cycles for any permutation S , we will again get cycles of lengths 10, 10, 2, 2, 1, 1. Therefore, if the plugboard settings are changed, but the initial positions of the rotors remain the same, then the cycle lengths remain unchanged.

You might have noticed that in the decomposition of AD , BE , and CF into cycles, each cycle length appears an even number of times. This is a general phenomenon.

For an explanation, see [Appendix E](#) of the aforementioned book by Kozaczuk.

Rejewski and his colleagues compiled a catalog of all 105456 initial settings of the rotors along with the set of cycle lengths for the corresponding three permutations AD , BE , CF . In this way, they could take the ciphertexts for a given day, deduce the cycle lengths, and find the small number of corresponding initial settings for the rotors. Each of these substitutions could be tried individually. The effect of the plugboard (when the correct setting was used) was then merely a substitution cipher, which was easily broken. This method worked until September 1938, when a modified method of transmitting message keys was adopted. Modifications of the above technique were again used to decrypt the messages. The process was also mechanized, using machines called “bombe” to find daily keys, each in around two hours.

These techniques were extended by the British at Bletchley Park during World War II and included building more sophisticated “bombe.” These machines, designed by Alan Turing, are often considered to have been the first electronic computers.

2.8 Exercises

1. Caesar wants to arrange a secret meeting with Marc Antony, either at the Tiber (the river) or at the Coliseum (the arena). He sends the ciphertext *EVIRE*. However, Antony does not know the key, so he tries all possibilities. Where will he meet Caesar? (Hint: This is a trick question.)
2. Show that each of the ciphertexts *ZOMCIH* and *ZKNGZR*, which were obtained by shift ciphers from one-word plaintexts, has two different decryptions.
3. The ciphertext *UCR* was encrypted using the affine function $9x + 2 \pmod{26}$. Find the plaintext.
4. The ciphertext *JLH* was obtained by affine encryption with the function $9x + 1 \pmod{26}$. Find the plaintext.
5. Encrypt *howareyou* using the affine function $5x + 7 \pmod{26}$. What is the decryption function? Check that it works.
6. You encrypt messages using the affine function $9x + 2 \pmod{26}$. Decrypt the ciphertext *GM*.
7. A child has learned about affine ciphers. The parent says *NONONO*. The child responds with *hahaha*, and quickly claims that this is a decryption of the parent's message. The parent asks for the encryption function. What answer should the child give?
8. You try to encrypt messages using the affine cipher $4x + 1 \pmod{26}$. Find two letters that encrypt to the same ciphertext letter.
9. The following ciphertext was encrypted by an affine cipher mod 26:
$$CRWWZ.$$
The plaintext starts *ha*. Decrypt the message.
10. Alice encrypts a message using the affine function $x \mapsto ax + b \pmod{26}$ for some a . The ciphertext is *FAP*. The third letter of the plaintext is *T*. Find the plaintext.
11. Suppose you encrypt using an affine cipher, then encrypt the encryption using another affine cipher (both are working mod 26). Is there any advantage to doing this, rather than using a single affine cipher? Why or why not?
12. Find all affine ciphers mod 26 for which the decryption function equals the encryption function. (There are 28 of them.)

13. Suppose we work mod 27 instead of mod 26 for affine ciphers.
 How many keys are possible? What if we work mod 29?
14. The ciphertext *XVASDW* was encrypted using an affine function $ax + 1 \pmod{26}$. Determine a and decrypt the message.
15. Suppose that you want to encrypt a message using an affine cipher. You let $a = 0, b = 1, \dots, z = 25$, but you also include $? = 26, ; = 27, " = 28, ! = 29$. Therefore, you use $x \mapsto \alpha x + \beta \pmod{30}$ for your encryption function, for some integers α and β .
1. Show that there are exactly eight possible choices for the integer α (that is, there are only eight choices of α (with $0 < \alpha < 30$) that allow you to decrypt).
 2. Suppose you try to use $\alpha = 10, \beta = 0$. Find two plaintext letters that encrypt to the same ciphertext letter.
16. You are trying to encrypt using the affine function $13x + 22 \pmod{26}$.
1. Encrypt *HATE* and *LOVE*. Why is decryption impossible?
 2. Find two different three-letter words that encrypt to *WWW*.
 3. Challenge: Find a word (that is legal in various word games) that encrypts to *JJJ*. (There are four such words.)
17. You want to carry out an affine encryption using the function $\alpha x + \beta$, but you have $\gcd(\alpha, 26) = d > 1$. Show that if $x_1 = x_2 + (26/d)$, then $\alpha x_1 + \beta \equiv \alpha x_2 + \beta \pmod{26}$. This shows that you will not be able to decrypt uniquely in this case.
18. You encrypt the message *zzzzzzzzzz* (there are 10 *z*'s) using the following cryptosystems:
1. affine cipher
 2. Vigenère cipher with key length 7
- Eve intercepts the ciphertexts. She knows the encryption methods (including key size) and knows what your plaintext is (she can hear you snoring). For each of the two cryptosystems, determine whether or not Eve can use this information to determine the key. Explain your answer.
19. Suppose there is a language that has only the letters *a* and *b*. The frequency of the letter *a* is .1 and the frequency of *b* is .9. A

message is encrypted using a Vigenère cipher (working mod 2 instead of mod 26). The ciphertext is BABABAAABA. The key length is 1, 2, or 3.

1. Show that the key length is probably 2.
 2. Using the information on the frequencies of the letters, determine the key and decrypt the message.
20. Suppose you have a language with only the three letters a , b , c , and they occur with frequencies .9, .09, and .01, respectively. The ciphertext *BCCCBBCBC* was encrypted by the Vigenère method (shifts are mod 3, not mod 26). Find the plaintext (Note: The plaintext is not a meaningful English message.)
21. Suppose you have a language with only the three letters a , b , c , and they occur with frequencies .7, .2, .1, respectively. The following ciphertext was encrypted by the Vigenère method (shifts are mod 3 instead of mod 26, of course):

ABCBABBAC.

Suppose you are told that the key length is 1, 2, or 3. Show that the key length is probably 2, and determine the most probable key.

22. Victor designs a cryptosystem (called “Vector”) as follows: He writes the letters in the plaintext as numbers mod 26 (with $a = 0$, $b = 1$, etc.) and groups them five at a time into five-dimensional vectors. His key is a five-dimensional vector. The encryption is adding the key vector mod 26 to each plaintext vector (so this is a shift cipher with vectors in place of individual letters).
1. Describe a chosen plaintext attack on this system. Give the *explicit* plaintext used and how you get the key from the information you obtain.
 2. Victor’s system is not new. It is the same as what well-known system?
23. If \mathbf{v} and \mathbf{w} are two vectors in n -dimensional space,
$$\mathbf{v} \cdot \mathbf{w} = |\mathbf{v}| |\mathbf{w}| \cos \theta,$$
 where θ is the angle between the two vectors (measured in the two-dimensional plane spanned by the two vectors), and $|\mathbf{v}|$ denotes the length of \mathbf{v} . Use this fact to show that, in the notation of Section 2.3, the dot product $\mathbf{A}_0 \cdot \mathbf{A}_i$ is largest when $i = 0$.
24. Alice uses an improvement of the Vigenère cipher. She chooses five affine functions

$$a_1x + b_1, a_2x + b_2, \dots, a_5x + b_5 \pmod{26}$$

and she uses these to encrypt in the style of Vigenère. Namely, she encrypts the first plaintext letter using $a_1x + b_1$, the second letter

using $a_2x + b_2$, etc.

1. What condition do a_1, a_2, \dots, a_5 need to satisfy for Bob (who knows the key) to able to decrypt the message?
 2. Describe how to do a *chosen plaintext* attack to find the key. Give the plaintext *explicitly* and explain how it yields the key. (Note: the solution has nothing to do with frequencies of letters.)
25. Alice is sending a message to Bob using one of the following cryptosystems. In fact, Alice is bored and her plaintext consists of the letter a repeated a few hundred times. Eve knows what system is being used, but not the key, and intercepts the ciphertext. For systems (a), (b), and (c), state how Eve will recognize that the plaintext is one repeated letter and decide whether or not Eve can deduce the letter and the key.
1. Shift cipher
 2. Affine cipher
 3. Vigenère cipher
26. The operator of a Vigenère encryption machine is bored and encrypts a plaintext consisting of the same letter of the alphabet repeated several hundred times. The key is a seven-letter English word. Eve knows that the key is a word but does not yet know its length.
1. What property of the ciphertext will make Eve suspect that the plaintext is one repeated letter and will allow her to guess that the key length is seven?
 2. Once Eve guesses that the plaintext is one repeated letter, how can she determine the key? (Hint: You need the fact that no English word of length seven is a shift of another English word.)
 3. Suppose Eve doesn't notice the property needed in part (a), and therefore uses the method of displacing then counting matches for finding the length of the key. What will the number of matches be for the various displacements? In other words, why will the length of the key become very obvious by this method?
27. Use the Playfair cipher with the keyword *Cryptography* to encrypt
Did he play fair at St Andrews golf course.
28. The ciphertext

BP EG FC AI MA MG PO KB HU

was encrypted using the Playfair cipher with keyword *Archimedes*. Find the plaintext.

29. Encrypt the plaintext *secret* using the ADFGX cipher with the 5×5 matrix in [Section 2.6](#) and the keyword *spy*.
30. The ciphertext *AAAAFXGGFAFFGGFGXAFGADGGAXXXFX* was encrypted using the ADFGX cipher with the 5×5 matrix in [Section 2.6](#) and the keyword *broken*. Find the plaintext.
31. Suppose Alice and Bob are using a cryptosystem with a 128-bit key, so there are 2^{128} possible keys. Eve is trying a brute-force attack on the system.
 1. Suppose it takes 1 day for Eve to try 2^{64} possible keys. At this rate, how long will it take for Eve to try all 2^{128} keys? (Hint: The answer is not 2 days.)
 2. Suppose Alice waits 10 years and then buys a computer that is 100 times faster than the one she now owns (so it takes only $1/100$ of a day, which is 864 seconds, to try 2^{64} keys). Will she finish trying all 2^{128} keys before or after what she does in part (a)? (Note: This is a case where Aesop's Fable about the Tortoise and the Hare has a different ending.)
32. In the mid-1980s, a recruiting advertisement for NSA had 1 followed by one hundred 0s at the top. The text began “You’re looking at a ‘googol.’ Ten raised to the 100th power. One followed by 100 zeroes. Counting 24 hours a day, you would need 120 years to reach a googol. Two lifetimes. It’s a number that’s impossible to grasp. A number beyond our imagination.”

How many numbers would you have to count each second in order to reach a googol in 120 years? (This problem is not related to the cryptosystems in this chapter. It is included to show how big 100-digit numbers are from a computational viewpoint. Regarding the ad, one guess is that the advertising firm assumed that the time it took to factor a 100-digit number back then was the same as the time it took to count to a googol.)

2.9 Computer Problems

1. The following ciphertext was encrypted by a shift cipher:

ycvejqwvhqtdtwvwu

Decrypt. (The ciphertext is stored in the downloadable computer files (bit.ly/2JbcS6p) under the name *ycve*.)

2. The following ciphertext was the output of a shift cipher:

lcllewljazlnnzmvyyiylhrmhza

By performing a frequency count, guess the key used in the cipher. Use the computer to test your hypothesis. What is the decrypted plaintext? (The ciphertext is stored in the downloadable computer files (bit.ly/2JbcS6p) under the name *lcll*.)

3. The following was encrypted by an affine cipher:

jidfbidzzteztxjsichfoihuszzfsaichbipahsibdhu
hzsichjujgfabbczggjsvzubehhgjsv. Decrypt it. (This quote (NYTimes, 12/7/2014) is by Mark Wahlberg from when he was observing college classes in order to play a professor in "The Gambler." The ciphertext is stored in the downloadable computer files (bit.ly/2JbcS6p) under the name *jidf*.) (Hint: The command "frequency" could be useful. The plaintext has 9 e's, 3 d's, and 3 w's.)

4. The following ciphertext was encrypted by an affine cipher:

edsgickxhuklzveqzvkwkzukcvuh

The first two letters of the plaintext are *if*. Decrypt. (The ciphertext is stored in the downloadable computer files (bit.ly/2JbcS6p) under the name *edsg*.)

5. The following ciphertext was encrypted by an affine cipher using the function $3x + b$ for some b :

tcabtiqmfheqqmrvmvtmaq

Decrypt. (The ciphertext is stored in the downloadable computer files (bit.ly/2JbcS6p) under the name *tcab*.)

6. Experiment with the affine cipher $y \equiv mx + n \pmod{26}$ for values of $m > 26$. In particular, determine whether or not these encryptions are the same as ones obtained with $m < 26$.
7. In this problem you are to get your hands dirty doing some programming. Write some code that creates a new alphabet $\{A, C, G, T\}$. For example, this alphabet could correspond to the four nucleotides adenine, cytosine, guanine, and thymine, which are the basic building blocks of DNA and RNA codes. Associate the letters A, C, G, T with the numbers 0, 1, 2, 3, respectively.
 1. Using the shift cipher with a shift of 1, encrypt the following sequence of nucleotides, which is taken from the beginning of the thirteenth human chromosome:

*GAATTCTGGCCGCAATTAACCCCTCACTAAAGGGATCT
CTAGAACT.*
 2. Write a program that performs affine ciphers on the nucleotide alphabet. What restrictions are there on the affine cipher?
8. The following was encrypted using by the Vigenère method using a key of length at most 6. Decrypt it and decide what is unusual about the plaintext. How did this affect the results?

hdsfgvmkoowafweetcmfthskucaqbilgjofmaqlgsp
vatvxqbiryscpcfr
mvsrvnqlszdmgaqsakmlupsqforvtwdfcjzvgso
aoqsacjkbrsevbel
vbksarlscdcaarmnvrysyxqgveellcyluwwveoafgc
lazowafojdjhssfi
ksepsoywxafowlbfcscocylngqsyzxgjbmlvgrrggokg
fgmhlmjeabsjvgml
nrvqzcrggcrghgeupcyfgtydycjkhqluhgxgzovqsw
pdvbwssffsenbxapa
sgazmyuhgsfhmftayjxmwznrsofrsoaopgauaaarmf
tqsmahvqecev

(The ciphertext is stored under the name *hdsf* in the downloadable computer files (bit.ly/2JbcS6p). The plaintext is from Gadsby

by Ernest Vincent Wright.)

9. The following was encrypted by the Vigenère method. Find the plaintext.

```
ocwyikoooniwugpmxwktzdwtssayjzwyemdlbnqaa
avsudvbrflauplo
oubfgqhgczcmgzlatoedcsdeidpbhtmuovpiekifpi
mfnoamvlpqfxejsm
xmpgkccaykwfzpyuavtelwhrhmwkbvgtguvtefjlo
dfefkvpgrsorvg
tajbsauhzralkwuowhgedefnswmrciwcpaaavogpd
nfpktbdbalsisurln
psjyeatcucesohhdarkhwotikbroqrdfmzghguceb
vgwdqxgpbgqwlpb
daylooqdmuhbdqgmyweuik
```

(The ciphertext is stored under the name *ocwy* in the downloadable computer files (bit.ly/2JbcS6p). The plaintext is from The Adventure of the Dancing Men by Sir Arthur Conan Doyle.)

10. The following was encrypted by the Vigenère method. Decrypt it.
(The ciphertext is stored under the name *xkju* in the downloadable computer files (bit.ly/2JbcS6p)).

```
xkjurowmlpxwznpimvbqjcnowxpcchhvvfvsl1fv
xhazityxohulxqoj
axelzxmyjaqfstsrulhhucdskbxknjqidallpqsl1
uhiaqfpbpcidsvci
hwhewthbtxr1jnrsncihuvffuxvoukj1jswmaqfvj
wjsdyljogjxdboxa
jultucpzmpliwmlobzxoodybafdsxgqfadshxnxe
hsaruojaqfpfkndh
saafvulluwtaqfrupwjrszxgpufutjqiynrnyntwmh
cukjfbirzsmehhsj
shyondzzntzmplilrwnmwmvlvuryonthuhabwnvw
```

Chapter 3 Basic Number Theory

In modern cryptographic systems, the messages are represented by numerical values prior to being encrypted and transmitted. The encryption processes are mathematical operations that turn the input numerical values into output numerical values. Building, analyzing, and attacking these cryptosystems requires mathematical tools. The most important of these is number theory, especially the theory of congruences. This chapter presents the basic tools needed for the rest of the book. More advanced topics such as factoring, discrete logarithms, and elliptic curves, will be treated in later chapters ([Chapters 9, 10, and 21](#), respectively).

3.1 Basic Notions

3.1.1 Divisibility

Number theory is concerned with the properties of the integers. One of the most important is divisibility.

Definition

Let a and b be integers with $a \neq 0$. We say that a **divides** b , if there is an integer k such that $b = ak$. This is denoted by $a|b$. Another way to express this is that b is a multiple of a .

Example

$3|15$, $-15|60$, $7 \nmid 18$ (does not divide).

The following properties of divisibility are useful.

Proposition

Let a, b, c represent integers.

1. For every $a \neq 0$, $a|0$ and $a|a$. Also, $1|b$ for every b .
2. If $a|b$ and $b|c$, then $a|c$.
3. If $a|b$ and $a|c$, then $a|(sb + tc)$ for all integers s and t .

Proof. Since $0 = a \cdot 0$, we may take $k = 0$ in the definition to obtain $a|0$. Since $a = a \cdot 1$, we take $k = 1$

to prove $a|a$. Since $b = 1 \cdot b$, we have $1|b$. This proves (1). In (2), there exist k and ℓ such that $b = ak$ and $c = b\ell$. Therefore, $c = (k\ell)a$, so $a|c$. For (3), write $b = ak_1$ and $c = ak_2$. Then $sb + tc = a(sk_1 + tk_2)$, so $a|sb + tc$.

For example, take $a = 2$ in part (2). Then $2|b$ simply means that b is even. The statement in the proposition says that c , which is a multiple of the even number b , must also be even (that is, a multiple of $a = 2$).

3.1.2 Prime Numbers

A number $p > 1$ whose positive divisors are only 1 and itself is called a **prime number**. The first few primes are 2, 3, 5, 7, 11, 13, 17, \dots . An integer $n > 1$ that is not prime is called **composite**, which means that n must be expressible as a product ab of integers with $1 < a, b < n$. A fact, known already to Euclid, is that there are infinitely many prime numbers. A more precise statement is the following, proved in 1896.

Prime Number Theorem

Let $\pi(x)$ be the number of primes less than x . Then

$$\pi(x) \approx \frac{x}{\ln x},$$

in the sense that the ratio $\pi(x)/(x/\ln x) \rightarrow 1$ as $x \rightarrow \infty$.

We won't prove this here; its proof would lead us too far away from our cryptographic goals. In various applications, we'll need large primes, say of around 300 digits. We can estimate the number of 300-digit primes as follows:

$$\pi(10^{300}) - \pi(10^{299}) \approx \frac{10^{300}}{\ln 10^{300}} - \frac{10^{299}}{\ln 10^{299}} \approx 1.4 \times 10^{297}.$$

So there are certainly enough such primes. Later, we'll discuss how to find them.

Prime numbers are the building blocks of the integers. Every positive integer has a unique representation as a product of prime numbers raised to different powers. For example, 504 and 1125 have the following factorizations:

$$504 = 2^3 3^2 7, \quad 1125 = 3^2 5^3.$$

Moreover, these factorizations are unique, except for reordering the factors. For example, if we factor 504 into primes, then we will always obtain three factors of 2, two factors of 3, and one factor of 7. Anyone who obtains the prime 41 as a factor has made a mistake.

Theorem

Every positive integer is a product of primes. This factorization into primes is unique, up to reordering the factors.

Proof. There is a small technicality that must be dealt with before we begin. When dealing with products, it is convenient to make the convention that an empty product equals 1. This is similar to the convention that $x^0 = 1$. Therefore, the positive integer 1 is a product of primes, namely the empty product. Also, each prime is regarded as a one-factor product of primes.

Suppose there exist positive integers that are not products of primes. Let n be the smallest such integer. Then n cannot be 1 (= the empty product), or a prime (= a one-factor product), so n must be composite. Therefore, $n = ab$ with $1 < a, b < n$. Since n is the smallest positive integer that is not a product of primes, both a and b are products of primes. But a product of

primes times a product of primes is a product of primes, so $n = ab$ is a product of primes. This contradiction shows that the set of integers that are not products of primes must be the empty set. Therefore, every positive integer is a product of primes.

The uniqueness of the factorization is more difficult to prove. We need the following very important property of primes.

Lemma

If p is a prime and p divides a product of integers ab , then either $p|a$ or $p|b$. More generally, if a prime p divides a product $ab \cdots z$, then p must divide one of the factors a, b, \dots, z .

For example, when $p = 2$, this says that if a product of two integers is even then one of the two integers must be even. The proof of the lemma will be given at the end of the next section, after we discuss the Extended Euclidean algorithm.

Continuing with the proof of the theorem, suppose that an integer n can be written as a product of primes in two different ways:

$$n = p_1^{a_1} p_2^{a_2} \cdots p_s^{a_s} = q_1^{b_1} q_2^{b_2} \cdots q_t^{b_t},$$

where p_1, \dots, p_s and q_1, \dots, q_t are primes, and the exponents a_i and b_j are nonzero. If a prime occurs in both factorizations, divide both sides by it to obtain a shorter relation. Continuing in this way, we may assume that none of the primes p_1, \dots, p_s occur among the q_j 's. Take a prime that occurs on the left side, say p_1 . Since p_1 divides n , which equals $q_1 q_2 \cdots q_t$, the lemma says that p_1 must divide one of the factors q_j . Since q_j is prime, $p_1 = q_j$. This contradicts the assumption that p_1 does not occur among the q_j 's.

Therefore, an integer cannot have two distinct factorizations, as claimed.

3.1.3 Greatest Common Divisor

The **greatest common divisor** of a and b is the largest positive integer dividing both a and b and is denoted by either $\gcd(a, b)$ or by (a, b) . In this book, we use the first notation. To avoid technicalities, we always assume implicitly that at least one of a and b is nonzero.

Example

$$\gcd(6, 4) = 2, \quad \gcd(5, 7) = 1, \quad \gcd(24, 60) = 12.$$

We say that a and b are **relatively prime** if $\gcd(a, b) = 1$. There are two standard ways for finding the gcd:

1. If you can factor a and b into primes, do so. For each prime number, look at the powers that it appears in the factorizations of a and b . Take the smaller of the two. Put these prime powers together to get the gcd. This is easiest to understand by examples:

$$576 = 2^6 3^2, \quad 135 = 3^3 5, \quad \gcd(576, 135) = 3^2 = 9$$
$$\gcd(2^5 3^4 7^2, 2^2 5^3 7) = 2^2 3^0 5^0 7^1 = 2^2 7 = 28.$$

Note that if a prime does not appear in a factorization, then it cannot appear in the gcd.

2. Suppose a and b are large numbers, so it might not be easy to factor them. The gcd can be calculated by a procedure known as the **Euclidean algorithm**. It goes back to what everyone learned in grade school: division with remainder. Before giving a formal description of the algorithm, let's see some examples.

Example

Compute $\gcd(482, 1180)$.

SOLUTION

Divide 482 into 1180. The quotient is 2 and the remainder is 216. Now divide the remainder 216 into 482. The quotient is 2 and the remainder is 50. Divide the remainder 50 into the previous remainder 216. The quotient is 4 and the remainder is 16. Continue this process of dividing the most recent remainder into the previous one. The last nonzero remainder is the gcd, which is 2 in this case:

$$\begin{aligned} 1180 &= 2 \cdot 482 + 216 \\ 482 &= 2 \cdot 216 + 50 \\ 216 &= 4 \cdot 50 + 16 \\ 50 &= 3 \cdot 16 + 2 \\ 16 &= 8 \cdot 2 + 0. \end{aligned}$$

Notice how the numbers are shifted:

remainder → to divisor → to dividend → to ignore.

Here is another example:

$$\begin{aligned} 12345 &= 1 \cdot 11111 + 1234 \\ 11111 &= 9 \cdot 1234 + 5 \\ 1234 &= 246 \cdot 5 + 4 \\ 5 &= 1 \cdot 4 + 1 \\ 4 &= 4 \cdot 1 + 0. \end{aligned}$$

Therefore, $\gcd(12345, 11111) = 1$.

Using these examples as guidelines, we can now give a more formal description of the **Euclidean algorithm**. Suppose that a is greater than b . If not, switch a and b . The first step is to divide a by b , hence represent a in the form

$$a = q_1 b + r_1.$$

If $r_1 = 0$, then b divides a and the greatest common divisor is b . If $r_1 \neq 0$, then continue by representing b in the form

$$b = q_2 r_1 + r_2.$$

Continue in this way until the remainder is zero, giving the following sequence of steps:

$$\begin{aligned} a &= q_1 b + r_1 \\ b &= q_2 r_1 + r_2 \\ r_1 &= q_3 r_2 + r_3 \\ &\vdots \quad \vdots \quad \vdots \\ r_{k-2} &= q_k r_{k-1} + r_k \\ r_{k-1} &= q_{k+1} r_k. \end{aligned}$$

The conclusion is that

$$\gcd(a, b) = r_k.$$

There are two important aspects to this algorithm:

1. It does not require factorization of the numbers.
2. It is fast.

For a proof that it actually computes the gcd, see

[Exercise 59](#).

3.2 The Extended Euclidean Algorithm

The Euclidean Algorithm computes greatest common divisors quickly, but also, with only slightly more work, yields a very useful fact: $\gcd(a, b)$ can be expressed as a linear combination of a and b . That is, there exist integers x and y such that $\gcd(a, b) = ax + by$. For example,

$$\begin{aligned} 1 &= \gcd(45, 13) = 45 \cdot (-2) + 13 \cdot 7 \\ 7 &= \gcd(259, 119) = 259 \cdot 6 - 119 \cdot 13. \end{aligned}$$

The **Extended Euclidean Algorithm** will tell us how to find x and y . Rather than give a set of equations, we'll show how it works with the two examples we calculated in Subsection 3.1.3.

When we computed $\gcd(12345, 11111)$, we did the following calculation:

$$\begin{aligned} 12345 &= 1 \cdot 11111 + 1234 \\ 11111 &= 9 \cdot 1234 + 5 \\ 1234 &= 246 \cdot 5 + 4 \\ 5 &= 1 \cdot 4 + 1. \end{aligned}$$

For the Extended Euclidean Algorithm, we'll form a table with three columns and explain how they arise as we compute them.

We begin by forming two rows and three columns. The first entries in the rows are the original numbers we started with, namely 12345 and 11111. We will do some calculations so that we always have

$$\text{entry in first column} = 12345x + 11111y,$$

where x and y are integers. The first two lines are trivial: $12345 = 1 \cdot 12345 + 0 \cdot 11111$ and

$$11111 = 0 \cdot 12345 + 1 \cdot 11111:$$

x	y
12345	1 0
11111	0 1

The first line in our gcd (12345, 11111) calculation tells us that $12345 = 1 \cdot 11111 + 1234$. We rewrite this as $1234 = 12345 - 1 \cdot 11111$. Using this, we compute

$$(1\text{st row}) - 1 \cdot (2\text{nd row}),$$

yielding the following:

x	y
12345	1 0
11111	0 1
1234	1 -1 (1st row) - 1 · (2nd row).

In effect, we have done the following subtraction:

$$\begin{aligned} 12345 &= 12345(1) + 11111(0) \\ 11111 &= 12345(0) + 11111(1) \\ 1234 &= 12345(0) + 11111(-1). \end{aligned}$$

Therefore, the last line tells us that

$$1234 = 12345 \cdot 1 + 11111 \cdot (-1).$$

We now move to the second row of our gcd calculation. This says that $11111 = 9 \cdot 1234 + 5$, which we rewrite as $5 = 11111 - 9 \cdot 1234$. This tells us to compute $(2\text{nd row}) - 9 \cdot (3\text{rd row})$. We write this as

x	y	
12345	1	0
11111	0	1
1234	1	-1
5	-9	10
		(2nd row) - 9 · (3rd row).

The last line tells us that

$$5 = 12345 \cdot (-9) + 11111 \cdot 10.$$

The third row of our gcd calculation tells us that

$$4 = 1234 - 246 \cdot 5. \text{ This becomes}$$

x	y	
12345	1	0
11111	0	1
1234	1	-1
5	-9	10
4	2215	-2461
		(3rd row) - 246 · (4th row).

Finally, we obtain

12345	1	0
11111	0	1
1234	1	-1
5	-9	10
4	2215	-2461

1	−2224	2471	(4th row) − (5th row).
---	-------	------	------------------------

This tells us that

$$1 = 12345 \cdot (-2224) + 11111 \cdot 2471.$$

Notice that as we proceeded, we were doing the Euclidean Algorithm in the first column. The first entry of each row is a remainder from the gcd calculation, and the entries in the second and third columns allow us to express the number in the first column as a linear combination of 12345 and 11111. The quotients in the Euclidean Algorithm tell us what to multiply a row by before subtracting it from the previous row.

Let's do another example using 482 and 1180 and our previous calculation that $\gcd(1180, 482) = 2$:

	x	y	
1180	1	0	
482	0	1	
216	1	−2	(1st row) − 2·(2nd row)
50	−2	5	(2nd row) − 2·(3rd row)
16	9	−22	(3rd row) − 4·(4th row)
2	−29	71	(4rd row) − 3·(5th row).

The end result is $2 = 1180 \cdot (-29) + 482 \cdot 71$.

To summarize, we state the following.

Theorem

Let a and b be integers with at least one of a, b nonzero. There exist integers x and y , which can be found by the Extended Euclidean Algorithm, such that

$$\gcd(a, b) = ax + by.$$

As a corollary, we deduce the lemma we needed during the proof of the uniqueness of factorization into primes.

Corollary

If p is a prime and p divides a product of integers ab , then either $p|a$ or $p|b$. More generally, if a prime p divides a product $ab \cdots z$, then p must divide one of the factors a, b, \dots, z .

Proof. First, let's work with the case $p|ab$. If p divides a , we are done. Now assume $p \nmid a$. We claim $p|b$. Since p is prime, $\gcd(a, p) = 1$ or p . Since $p \nmid a$, the gcd cannot be p . Therefore, $\gcd(a, p) = 1$, so there exist integers x, y with $ax + py = 1$. Multiply by b to obtain $abx + pby = b$. Since $p|ab$ and $p|p$, we have $p|abx + pby$, so $p|b$, as claimed.

If $p|ab \cdots z$, then $p|a$ or $p|b \cdots z$. If $p|a$, we're done. Otherwise, $p|b \cdots z$. We now have a shorter product. Either $p|b$, in which case we're done, or p divides the product of the remaining factors. Continuing in this way, we eventually find that p divides one of the factors of the product.

The property of primes stated in the corollary holds only for primes. For example, if we know a product ab is divisible by 6, we cannot conclude that a or b is a multiple of 6. The problem is that $6 = 2 \cdot 3$, and the 2 could be in a while the 3 could be in b , as seen in the example $60 = 4 \cdot 15$. More generally, if $n = ab$ is any composite, then $n|ab$ but $n \nmid a$ and $n \nmid b$. Therefore, the

primes, and 1, are the only integers with the property of the corollary.

3.3 Congruences

One of the most basic and useful notions in number theory is modular arithmetic, or congruences.

Definition

Let a, b, n be integers with $n \neq 0$. We say that

$$a \equiv b \pmod{n}$$

(read: a is **congruent** to b mod n) if $a - b$ is a multiple (positive or negative or zero) of n .

Another formulation is that $a \equiv b \pmod{n}$ if a and b differ by a multiple of n . This can be rewritten as
 $a = b + nk$ for some integer k (positive or negative).

Example

$$32 \equiv 7 \pmod{5}, \quad -12 \equiv 37 \pmod{7}, \quad 17 \equiv 17 \pmod{13}.$$

Note: Many computer programs regard $17 \pmod{10}$ as equal to the number 7, namely, the remainder obtained when 17 is divided by 10 (often written as $17 \% 10 = 7$). The notion of congruence we use is closely related. We have that two numbers are congruent mod n if they yield the same remainders when divided by n . For example, $17 \equiv 37 \pmod{10}$ because $17 \% 10$ and $37 \% 10$ are equal.

Congruence behaves very much like equality. In fact, the notation for congruence was intentionally chosen to resemble the notation for equality.

Proposition

Let a, b, c, n be integers with $n \neq 0$.

1. $a \equiv 0 \pmod{n}$ if and only if $n|a$.
2. $a \equiv a \pmod{n}$.
3. $a \equiv b \pmod{n}$ if and only if $b \equiv a \pmod{n}$.
4. If $a \equiv b$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$.

Proof. In (1), $a \equiv 0 \pmod{n}$ means that $a = a - 0$ is a multiple of n , which is the same as $n|a$. In (2), we have $a - a = 0 \cdot n$, so $a \equiv a \pmod{n}$. In (3), if $a \equiv b \pmod{n}$, write $a - b = nk$. Then $b - a = n(-k)$, so $b \equiv a \pmod{n}$. Reversing the roles of a and b gives the reverse implication. For (4), write $a = b + nk$ and $b = c + n\ell$. Then $a - c = n(k + \ell)$, so $a \equiv c \pmod{n}$.

Usually, we have $n > 0$ and we work with the integers mod n , denoted \mathbf{Z}_n . These may be regarded as the set $\{0, 1, 2, \dots, n-1\}$, with addition, subtraction, and multiplication mod n . If a is any integer, we may divide a by n and obtain a remainder in this set:

$$a = nq + r \text{ with } 0 \leq r < n.$$

(This is just division with remainder; q is the quotient and r is the remainder.) Then $a \equiv r \pmod{n}$, so every number a is congruent mod n to some integer r with $0 \leq r < n$.

Proposition

Let a, b, c, d, n be integers with $n \neq 0$, and suppose $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$. Then

$$a + c \equiv b + d, \quad a - c \equiv b - d, \quad ac \equiv bd \pmod{n}.$$

Proof. Write $a = b + nk$ and $c = d + n\ell$, for integers k and ℓ . Then $a + c = b + d + n(k + \ell)$, so $a + c \equiv b + d \pmod{n}$. The proof that $a - c \equiv b - d$ is similar. For multiplication, we have $ac = bd + n(dk + bl + nk\ell)$, so $ac \equiv bd$.

The proposition says you can perform the usual arithmetic operations of addition, subtraction, and multiplication with congruences. You must be careful, however, when trying to perform division, as we'll see.

If we take two numbers and want to multiply them modulo n , we start by multiplying them as integers. If the product is less than n , we stop. If the product is larger than $n - 1$, we divide by n and take the remainder. Addition and subtraction are done similarly. For example, the integers modulo 6 have the following addition table:

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

[3.3-1 Full Alternative Text](#)

A table for multiplication mod 6 is

\times	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

[3.3-2 Full Alternative Text](#)

Example

Here is an example of how we can do algebra mod n .

Consider the following problem: Solve

$$x + 7 \equiv 3 \pmod{17}.$$

SOLUTION

$$x \equiv 3 - 7 \equiv -4 \equiv 13 \pmod{17}.$$

There is nothing wrong with negative answers, but usually we write the final answer as an integer from 0 to $n - 1$ when we are working mod n .

3.3.1 Division

Division is much trickier mod n than it is with rational numbers. The general rule is that you can divide by a (mod n) when $\gcd(a, n) = 1$.

Proposition

Let a, b, c, n be integers with $n \neq 0$ and with $\gcd(a, n) = 1$. If $ab \equiv ac \pmod{n}$, then $b \equiv c \pmod{n}$. In other words, if a and n are relatively prime, we can divide both sides of the congruence by a .

Proof Since $\gcd(a, n) = 1$, there exist integers x, y such that $ax + ny = 1$. Multiply by $b - c$ to obtain

$$(ab - ac)x + n(b - c)y = b - c.$$

Since $ab - ac$ is a multiple of n , by assumption, and $n(b - c)y$ is also a multiple of n , we find that $b - c$ is a multiple of n . This means that $b \equiv c \pmod{n}$.

Example

Solve: $2x + 7 \equiv 3 \pmod{17}$.

SOLUTION

$2x \equiv 3 - 7 \equiv -4$, so $x \equiv -2 \equiv 15 \pmod{17}$. The division by 2 is allowed since $\gcd(2, 17) = 1$.

Example

Solve: $5x + 6 \equiv 13 \pmod{11}$.

SOLUTION

$5x \equiv 7 \pmod{11}$. Now what do we do? We want to divide by 5, but what does $7/5$ mean mod 11? Note that $7 \equiv 18 \equiv 29 \equiv 40 \equiv \dots \pmod{11}$. So $5x \equiv 7$ is the same as $5x \equiv 40$. Now we can divide by 5 and obtain $x \equiv 8 \pmod{11}$ as the answer. Note that $7 \equiv 8 \cdot 5 \pmod{11}$, so 8 acts like $7/5$.

The last example can be done another way. Since $5 \cdot 9 \equiv 1 \pmod{11}$, we see that 9 is the multiplicative inverse of 5 ($\pmod{11}$). Therefore, dividing by 5 can be accomplished by multiplying by 9. If we want to solve $5x \equiv 7 \pmod{11}$, we multiply both sides by 9 and obtain

$$x \equiv 45x \equiv 63 \equiv 8 \pmod{11}.$$

Proposition

Suppose $\gcd(a, n) = 1$. Let s and t be integers such that $as + nt = 1$ (they can be found using the extended Euclidean algorithm). Then $as \equiv 1 \pmod{n}$, so s is the multiplicative inverse for a (\pmod{n}).

Proof. Since $as - 1 = -nt$, we see that $as - 1$ is a multiple of n .

Notation: We let a^{-1} denote this s , so a^{-1} satisfies $a^{-1}a \equiv 1 \pmod{n}$.

The extended Euclidean algorithm is fairly efficient for computing the multiplicative inverse of a by the method stated in the proposition.

Example

Solve $11111x \equiv 4 \pmod{12345}$.

SOLUTION

In Section 3.2, from the calculation of $\gcd(12345, 11111)$ we obtained

$$1 = 12345 \cdot (-2224) + 11111 \cdot 2471.$$

This says that

$$11111 \cdot 2471 \equiv 1 \pmod{12345}.$$

Multiplying both sides of the original congruence by 2471 yields

$$x \equiv 9884 \pmod{12345}.$$

In practice, this means that if we are working mod 12345 and we encounter the fraction 4/11111, we can replace it with 9884. This might seem a little strange, but think about what 4/11111 means. It's simply a symbol to represent a quantity that, when multiplied by 11111, yields 4. When we are working mod 12345, the number 9884 also has this property since $11111 \times 9884 \equiv 4 \pmod{12345}$.

Let's summarize some of the discussion:

Finding

$$a^{-1} \pmod{n}$$

1. Use the extended Euclidean algorithm to find integers s and t such that $as + nt = 1$.
2. $a^{-1} \equiv s \pmod{n}$.

Solving

$$ax \equiv c \pmod{n} \text{ when } \gcd(a, n) = 1$$

(Equivalently, you could be working mod n and encounter a fraction c/a with $\gcd(a, n) = 1$.)

1. Use the extended Euclidean algorithm to find integers s and t such that $as + nt = 1$.
2. The solution is $x \equiv cs \pmod{n}$ (equivalently, replace the fraction c/a with $cs \pmod{n}$).

What if

$$\gcd(a, n) > 1?$$

Occasionally we will need to solve congruences of the form $ax \equiv b \pmod{n}$ when $\gcd(a, n) = d > 1$. The procedure is as follows:

1. If d does not divide b , there is no solution.
2. Assume $d|b$. Consider the new congruence

$$(a/d)x \equiv b/d \pmod{n/d}.$$

Note that $a/d, b/d, n/d$ are integers and $\gcd(a/d, n/d) = 1$. Solve this congruence by the above procedure to obtain a solution x_0 .

3. The solutions of the original congruence $ax \equiv b \pmod{n}$ are

$$x_0, \quad x_0 + (n/d), \quad x_0 + 2(n/d), \quad \dots, \quad x_0 + (d-1)(n/d) \pmod{n}.$$

Example

Solve $12x \equiv 21 \pmod{39}$.

SOLUTION

$\gcd(12, 39) = 3$, which divides 21. Divide by 3 to obtain the new congruence $4x \equiv 7 \pmod{13}$. A solution $x_0 = 5$ can be obtained by trying a few numbers, or by using the extended Euclidean algorithm. The solutions to the original congruence are $x \equiv 5, 18, 31 \pmod{39}$.

The preceding congruences contained x to the first power. However, nonlinear congruences are also useful. In several places in this book, we will meet equations of the form

$$x^2 \equiv a \pmod{n}.$$

First, consider $x^2 \equiv 1 \pmod{7}$. The solutions are $x \equiv 1, 6 \pmod{7}$, as we can see by trying the values

$0, 1, 2, \dots, 6$ for x . In general, when p is an odd prime, $x^2 \equiv 1 \pmod{p}$ has exactly the two solutions $x \equiv \pm 1 \pmod{p}$ (see Exercise 15).

Now consider $x^2 \equiv 1 \pmod{15}$. If we try the numbers $0, 1, 2, \dots, 14$ for x , we find that $x = 1, 4, 11, 14$ are solutions. For example, $11^2 \equiv 121 \equiv 1 \pmod{15}$.

Therefore, a quadratic congruence for a composite modulus can have more than two solutions, in contrast to the fact that a quadratic equation with real numbers, for example, can have at most two solutions. In Section 3.4, we'll discuss this phenomenon. In Chapters 9 (factoring), 18 (flipping coins), and 19 (identification schemes), we'll meet applications of this fact.

3.3.2 Working with Fractions

In many situations, it will be convenient to work with fractions mod n . For example, $1/2 \pmod{12345}$ is easier to write than $6173 \pmod{12345}$ (note that $2 \times 6173 \equiv 1 \pmod{12345}$). The general rule is that a fraction b/a can be used mod n if $\gcd(a, n) = 1$. Of course, it should be remembered that $b/a \pmod{n}$ really means $a^{-1}b \pmod{n}$, where a^{-1} denotes the integer mod n that satisfies $a^{-1}a \equiv 1 \pmod{n}$. But nothing will go wrong if it is treated as a fraction.

Another way to look at this is the following. The symbol “ $1/2$ ” is simply a symbol with exactly one property: If you multiply $1/2$ by 2, you get 1. In all calculations involving the symbol $1/2$, this is the only property that is used.

When we are working mod 12345, the number 6173 also has this property, since $6173 \times 2 \equiv 1 \pmod{12345}$. Therefore, $1/2 \pmod{12345}$ and $6173 \pmod{12345}$ may be used interchangeably.

Why can't we use fractions with arbitrary denominators? Of course, we cannot use $1/6 \pmod{6}$, since that

would mean dividing by $0 \pmod{6}$. But even if we try to work with $1/2 \pmod{6}$, we run into trouble. For example, $2 \equiv 8 \pmod{6}$, but we cannot multiply both sides by $1/2$, since $1 \not\equiv 4 \pmod{6}$. The problem is that $\gcd(2, 6) = 2 \neq 1$. Since 2 is a factor of 6, we can think of dividing by 2 as “partially dividing by 0.” In any case, it is not allowed.

3.4 The Chinese Remainder Theorem

In many situations, it is useful to break a congruence mod n into a system of congruences mod factors of n .

Consider the following example. Suppose we know that a number x satisfies $x \equiv 25 \pmod{42}$. This means that we can write $x = 25 + 42k$ for some integer k .

Rewriting 42 as $7 \cdot 6$, we obtain $x = 25 + 7(6k)$, which implies that $x \equiv 25 \equiv 4 \pmod{7}$. Similarly, since $x = 25 + 6(7k)$, we have $x \equiv 25 \equiv 1 \pmod{6}$.

Therefore,

$$x \equiv 25 \pmod{42} \Rightarrow \begin{cases} x \equiv 4 \pmod{7} \\ x \equiv 1 \pmod{6}. \end{cases}$$

The Chinese remainder theorem shows that this process can be reversed; namely, a system of congruences can be replaced by a single congruence under certain conditions.

Chinese Remainder Theorem

Suppose $\gcd(m, n) = 1$. Given integers a and b , there exists exactly one solution $x \pmod{mn}$ to the simultaneous congruences

$$x \equiv a \pmod{m}, \quad x \equiv b \pmod{n}.$$

Proof. There exist integers s, t such that $ms + nt = 1$. Then $ms \equiv 1 \pmod{n}$ and $nt \equiv 1 \pmod{m}$. Let $x = bms + ant$. Then $x \equiv ant \equiv a \pmod{m}$, and $x \equiv bms \equiv b \pmod{n}$, so a solution x exists. Suppose x_1 is another solution. Then $x \equiv x_1 \pmod{m}$ and $x \equiv x_1 \pmod{n}$, so $x - x_1$ is a multiple of both m and n .

Lemma

Let m, n be integers with $\gcd(m, n) = 1$. If an integer c is a multiple of both m and n , then c is a multiple of mn .

Proof. Let $c = mk = n\ell$. Write $ms + nt = 1$ with integers s, t . Multiply by c to obtain

$$c = cms + cnt = mn\ell s + mnkt = mn(\ell s + kt).$$

To finish the proof of the theorem, let $c = x - x_1$ in the lemma to find that $x - x_1$ is a multiple of mn .

Therefore, $x \equiv x_1 \pmod{mn}$. This means that any two solutions x to the system of congruences are congruent mod mn , as claimed.

Example

Solve $x \equiv 3 \pmod{7}$, $x \equiv 5 \pmod{15}$.

SOLUTION

$x \equiv 80 \pmod{105}$ (note: $105 = 7 \cdot 15$). Since $80 \equiv 3 \pmod{7}$ and $80 \equiv 5 \pmod{15}$, 80 is a solution. The theorem guarantees that such a solution exists, and says that it is uniquely determined mod the product mn , which is 105 in the present example.

How does one find the solution? One way, which works with small numbers m and n , is to list the numbers congruent to $b \pmod{n}$ until you find one that is congruent to $a \pmod{m}$. For example, the numbers congruent to $5 \pmod{15}$ are

$$5, 20, 35, 50, 65, 80, 95, \dots$$

Mod 7, these are $5, 6, 0, 1, 2, 3, 4, \dots$. Since we want $3 \pmod{7}$, we choose 80 .

For slightly larger numbers m and n , making a list would be inefficient. However, the proof of the theorem gives a fast method for finding x :

1. Use the Extended Euclidean algorithm to find s and t with $ms + nt = 1$.
2. Let $x \equiv bms + ant \pmod{mn}$.

Example

Solve $x \equiv 7 \pmod{12345}$, $x \equiv 3 \pmod{11111}$.

SOLUTION

First, we know from our calculations in [Section 3.2](#) that

$$12345 \cdot (-2224) + 11111 \cdot 2471 = 1,$$

so $s = -2224$ and $t = 2471$. Therefore,

$$x \equiv 3 \cdot 12345 \cdot (-2224) + 7 \cdot 11111 \cdot 2471 \equiv 109821127 \pmod{(11111 \cdot 12345)}.$$

How do you use the Chinese remainder theorem? The main idea is that if you start with a congruence mod a composite number n , you can break it into simultaneous congruences mod each prime power factor of n , then recombine the resulting information to obtain an answer mod n . The advantage is that often it is easier to analyze congruences mod primes or mod prime powers than to work mod composite numbers.

Suppose you want to solve $x^2 \equiv 1 \pmod{35}$. Note that $35 = 5 \cdot 7$. We have

$$x^2 \equiv 1 \pmod{35} \Leftrightarrow \begin{cases} x^2 \equiv 1 \pmod{7} \\ x^2 \equiv 1 \pmod{5}. \end{cases}$$

Now, $x^2 \equiv 1 \pmod{5}$ has two solutions: $x \equiv \pm 1 \pmod{5}$. Also, $x^2 \equiv 1 \pmod{7}$ has two solutions: $x \equiv \pm 1 \pmod{7}$. We can put these together in four ways:

$$\begin{aligned}
x &\equiv 1 \pmod{5}, & x &\equiv 1 \pmod{7} \rightarrow x &\equiv 1 \pmod{35}, \\
x &\equiv 1 \pmod{5}, & x &\equiv -1 \pmod{7} \rightarrow x &\equiv 6 \pmod{35}, \\
x &\equiv -1 \pmod{5}, & x &\equiv 1 \pmod{7} \rightarrow x &\equiv 29 \pmod{35}, \\
x &\equiv -1 \pmod{5}, & x &\equiv -1 \pmod{7} \rightarrow x &\equiv 34 \pmod{35}.
\end{aligned}$$

So the solutions of $x^2 \equiv 1 \pmod{35}$ are

$$x \equiv 1, 6, 29, 34 \pmod{35}.$$

In general, if $n = p_1 p_2 \cdots p_r$ is the product of r distinct odd primes, then $x^2 \equiv 1 \pmod{n}$ has 2^r solutions. This is a consequence of the following.

Chinese Remainder Theorem (General Form)

Let m_1, \dots, m_k be integers with $\gcd(m_i, m_j) = 1$ whenever $i \neq j$. Given integers a_1, \dots, a_k , there exists exactly one solution $x \pmod{m_1 \cdots m_k}$ to the simultaneous congruences

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}, \quad \dots, \quad x \equiv a_k \pmod{m_k}.$$

For example, the theorem guarantees there is a solution to the simultaneous congruences

$$x \equiv 1 \pmod{11}, \quad x \equiv -1 \pmod{13}, \quad x \equiv 1 \pmod{17}.$$

In fact, $x \equiv 1871 \pmod{11 \cdot 13 \cdot 17}$ is the answer.

[Exercise 57](#) gives a method for computing the number x in the theorem.

3.5 Modular Exponentiation

Throughout this book, we will be interested in numbers of the form

$$x^a \pmod{n}.$$

In this and the next couple of sections, we discuss some properties of numbers raised to a power modulo an integer.

Suppose we want to compute $2^{1234} \pmod{789}$. If we first compute 2^{1234} , then reduce mod 789, we'll be working with very large numbers, even though the final answer has only 3 digits. We should therefore perform each multiplication and then calculate the remainder. Calculating the consecutive powers of 2 would require that we perform the modular multiplication 1233 times. This method is too slow to be practical, especially when the exponent becomes very large. A more efficient way is the following (all congruences are mod 789).

We start with $2^2 \equiv 4 \pmod{789}$ and repeatedly square both sides to obtain the following congruences:

$$\begin{aligned} 2^4 &\equiv 4^2 \equiv 16 \\ 2^8 &\equiv 16^2 \equiv 256 \\ 2^{16} &\equiv 256^2 \equiv 49 \\ 2^{32} &\equiv 34 \\ 2^{64} &\equiv 367 \\ 2^{128} &\equiv 559 \\ 2^{256} &\equiv 37 \\ 2^{512} &\equiv 580 \\ 2^{1024} &\equiv 286. \end{aligned}$$

Since $1234 = 1024 + 128 + 64 + 16 + 2$ (this just means that 1234 equals 10011010010 in binary), we have

$$2^{1234} \equiv 286 \cdot 559 \cdot 367 \cdot 49 \cdot 4 \equiv 481 \pmod{789}.$$

Note that we never needed to work with a number larger than 788^2 .

The same method works in general. If we want to compute $a^b \pmod{n}$, we can do it with at most $2 \log_2(b)$ multiplications mod n , and we never have to work with numbers larger than n^2 . This means that exponentiation can be accomplished quickly, and not much memory is needed.

This method is very useful if a, b, n are 100-digit numbers. If we simply computed a^b , then reduced mod n , the computer's memory would overflow: The number a^b has more than 10^{100} digits, which is more digits than there are particles in the universe. However, the computation of $a^b \pmod{n}$ can be accomplished in fewer than 700 steps by the present method, never using a number of more than 200 digits.

Algorithmic versions of this procedure are given in **Exercise 56**. For more examples, see Examples 8 and 24–30 in the Computer Appendices.

3.6 Fermat's Theorem and Euler's Theorem

Two of the most basic results in number theory are Fermat's and Euler's theorems. Originally admired for their theoretical value, they have more recently proved to have important cryptographic applications and will be used repeatedly throughout this book.

Fermat's Theorem

If p is a prime and p does not divide a , then

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof. Let

$$S = \{1, 2, 3, \dots, p-1\}.$$

Consider the map $\psi : S \rightarrow S$ defined by $\psi(x) = ax \pmod{p}$. For example, when $p = 7$ and $a = 2$, the map ψ takes a number x , multiplies it by 2, then reduces the result mod 7.

We need to check that if $x \in S$, then $\psi(x)$ is actually in S ; that is, $\psi(x) \neq 0$. Suppose $\psi(x) = 0$. Then $ax \equiv 0 \pmod{p}$. Since $\gcd(a, p) = 1$, we can divide this congruence by a to obtain $x \equiv 0 \pmod{p}$, so $x \notin S$. This contradiction means that $\psi(x)$ cannot be 0, hence $\psi(x) \in S$. Now suppose there are $x, y \in S$ with $\psi(x) = \psi(y)$. This means $ax \equiv ay \pmod{p}$. Since $\gcd(a, p) = 1$, we can divide this congruence by a to obtain $x \equiv y \pmod{p}$. We conclude that if x, y are distinct elements of S , then $\psi(x)$ and $\psi(y)$ are distinct. Therefore,

$$\psi(1), \psi(2), \psi(3), \dots, \psi(p-1)$$

are distinct elements of S . Since S has only $p - 1$ elements, these must be the elements of S written in a some order. It follows that

$$\begin{aligned} & 1 \cdot 2 \cdot 3 \cdots (p-1) \\ & \equiv \psi(1) \cdot \psi(2) \cdot \psi(3) \cdots \psi(p-1) \\ & \equiv (a \cdot 1)(a \cdot 2)(a \cdot 3) \cdots (a \cdot (p-1)) \\ & \equiv a^{p-1}(1 \cdot 2 \cdot 3 \cdots (p-1)) \pmod{p}. \end{aligned}$$

Since $\gcd(j, p) = 1$ for $j \in S$, we can divide this congruence by 1, 2, 3, ..., $p - 1$. What remains is $1 \equiv a^{p-1} \pmod{p}$.

Example

$2^{10} = 1024 \equiv 1 \pmod{11}$. From this we can evaluate $2^{53} \pmod{11}$: Write $2^{53} = (2^{10})^5 2^3 \equiv 1^5 2^3 \equiv 8 \pmod{11}$. Note that when working mod 11, we are essentially working with the exponents mod 10, not mod 11. In other words, from $53 \equiv 3 \pmod{10}$, we deduce $2^{53} \equiv 2^3 \pmod{11}$.

The example leads us to a very important fact:

Basic Principle

Let p be prime and let a, x, y be integers with $\gcd(a, p) = 1$. If $x \equiv y \pmod{p-1}$, then $a^x \equiv a^y \pmod{p}$. In other words, if you want to work mod p , you should work mod $p - 1$ in the exponent.

Proof. Write $x = y + (p-1)k$. Then

$$a^x = a^{y+(p-1)k} = a^y (a^{p-1})^k \equiv a^y 1^k \equiv a^y \pmod{p}.$$

This completes the proof.

In the rest of this book, almost every time you see a congruence mod $p - 1$, it will involve numbers that appear in exponents. The Basic Principle that was just stated shows that this translates into an overall congruence mod p . Do not make the (unfortunately, very common) mistake of working mod p in the exponent with the hope that it will yield an overall congruence mod p . It doesn't.

We can often use Fermat's theorem to show that a number is composite, without factoring. For example, let's show that 49 is composite. We use the technique of [Section 3.5](#) to calculate

$$\begin{aligned} 2^2 &\equiv 4 \pmod{49} \\ 2^4 &\equiv 16 \\ 2^8 &\equiv 16^2 \equiv 11 \\ 2^{16} &\equiv 11^2 \equiv 23 \\ 2^{32} &\equiv 23^2 \equiv 39 \\ 2^{48} &\equiv 2^{32}2^{16} \equiv 39 \cdot 23 \equiv 15. \end{aligned}$$

Since

$$2^{48} \not\equiv 1 \pmod{49},$$

we conclude that 49 cannot be prime (otherwise, Fermat's theorem would require that $2^{48} \equiv 1 \pmod{49}$). Note that we showed that a factorization must exist, even though we didn't find the factors.

Usually, if $2^{n-1} \equiv 1 \pmod{n}$, the number n is prime. However, there are exceptions: $561 = 3 \cdot 11 \cdot 17$ is composite but $2^{560} \equiv 1 \pmod{561}$. We can see this as follows: Since $560 \equiv 0 \pmod{2}$, we have $2^{560} \equiv 2^0 \equiv 1 \pmod{3}$. Similarly, since $560 \equiv 0 \pmod{10}$ and $560 \equiv 0 \pmod{16}$, we can conclude that $2^{560} \equiv 1 \pmod{11}$ and $2^{560} \equiv 1 \pmod{17}$. Putting things together via the Chinese remainder theorem, we find that $2^{560} \equiv 1 \pmod{561}$.

Another such exception is $1729 = 7 \cdot 13 \cdot 19$. However, these exceptions are fairly rare in practice. Therefore, if $2^{n-1} \equiv 1 \pmod{n}$, it is quite likely that n is prime. Of course, if $2^{n-1} \not\equiv 1 \pmod{n}$, then n cannot be prime.

Since $2^{n-1} \pmod{n}$ can be evaluated very quickly (see [Section 3.5](#)), this gives a way to search for prime numbers. Namely, choose a starting point n_0 and successively test each odd number $n \geq n_0$ to see whether $2^{n-1} \equiv 1 \pmod{n}$. If n fails the test, discard it and proceed to the next n . When an n passes the test, use more sophisticated techniques (see [Section 9.3](#)) to test n for primality. The advantage is that this procedure is much faster than trying to factor each n , especially since it eliminates many n quickly. Of course, there are ways to speed up the search, for example, by first eliminating any n that has small prime factors.

For example, suppose we want to find a random 300-digit prime. Choose a random 300-digit odd integer n_0 as a starting point. Successively, for each odd integer $n \geq n_0$, compute $2^{n-1} \pmod{n}$ by the modular exponentiation technique of [Section 3.5](#). If $2^{n-1} \not\equiv 1 \pmod{n}$, Fermat's theorem guarantees that n is not prime. This will probably throw out all the composites encountered. When you find an n with $2^{n-1} \equiv 1 \pmod{n}$, you probably have a prime number. But how many n do we have to examine before finding the prime? The Prime Number Theorem (see [Subsection 3.1.2](#)) says that the number of 300-digit primes is approximately 1.4×10^{297} , so approximately 1 out of every 690 numbers is prime. But we are looking only at odd numbers, so we expect to find a prime approximately every 345 steps. Since the modular exponentiations can be done quickly, the whole process takes much less than a second on a laptop computer.

We'll also need the analog of Fermat's theorem for a composite modulus n . Let $\phi(n)$ be the number of integers $1 \leq a \leq n$ such that $\gcd(a, n) = 1$. For example, if $n = 10$, then there are four such integers, namely 1,3,7,9. Therefore, $\phi(10) = 4$. Often ϕ is called **Euler's ϕ -function**.

If p is a prime and $n = p^r$, then we must remove every p th number in order to get the list of a 's with $\gcd(a, n) = 1$, which yields

$$\phi(p^r) = (1 - \frac{1}{p})p^r.$$

In particular,

$$\phi(p) = p - 1.$$

More generally, it can be deduced from the Chinese remainder theorem that for any integer n ,

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right),$$

where the product is over the distinct primes p dividing n . When $n = pq$ is the product of two distinct primes, this yields

$$\phi(pq) = (p - 1)(q - 1).$$

Examples

$$\phi(10) = (2 - 1)(5 - 1) = 4,$$

$$\phi(120) = 120\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) = 32$$

Euler's Theorem

If $\gcd(a, n) = 1$, then

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

Proof. The proof of this theorem is almost the same as the one given for Fermat's theorem. Let S be the set of integers $1 \leq x \leq n$ with $\gcd(x, n) = 1$. Let $\psi : S \rightarrow S$ be defined by $\psi(x) \equiv ax \pmod{n}$. As in the proof of Fermat's theorem, the numbers $\psi(x)$ for $x \in S$ are the numbers in S written in some order. Therefore,

$$\prod_{x \in S} x \equiv \prod_{x \in S} \psi(x) \equiv a^{\phi(n)} \prod_{x \in S} x.$$

Dividing out the factors $x \in S$, we are left with $1 \equiv a^{\phi(n)} \pmod{n}$.

Note that when $n = p$ is prime, Euler's theorem is the same as Fermat's theorem.

Example

What are the last three digits of 7^{803} ?

SOLUTION

Knowing the last three digits is the same as working mod 1000. Since

$$\phi(1000) = 1000 \left(1 - \frac{1}{2}\right) \left(1 - \frac{1}{5}\right) = 400, \text{ we have } 7^{803} = (7^{400})^2 7^3 \equiv 7^3 \equiv 343 \pmod{1000}.$$

Therefore, the last three digits are 343.

In this example, we were able to change the exponent 803 to 3 because $803 \equiv 3 \pmod{\phi(1000)}$.

Example

Compute $2^{43210} \pmod{101}$.

SOLUTION

Note that 101 is prime. From Fermat's theorem, we know that $2^{100} \equiv 1 \pmod{101}$. Therefore,

$$2^{43210} \equiv (2^{100})^{432} 2^{10} \equiv 1^{432} 2^{10} \equiv 1024 \equiv 14 \pmod{101}.$$

In this case, we were able to change the exponent 43210 to 10 because $43210 \equiv 10 \pmod{100}$.

To summarize, we state the following, which is a generalization of what we know for primes:

Basic Principle

Let a, n, x, y be integers with $n \geq 1$ and $\gcd(a, n) = 1$. If $x \equiv y \pmod{\phi(n)}$, then $a^x \equiv a^y \pmod{n}$. In other words, if you want to work mod n , you should work mod $\phi(n)$ in the exponent.

Proof. Write $x = y + \phi(n)k$. Then

$$a^x = a^{y+\phi(n)k} = a^y (a^{\phi(n)})^k \equiv a^y 1^k \equiv a^y \pmod{n}.$$

This completes the proof.

This extremely important fact will be used repeatedly in the remainder of the book. Review the preceding examples until you are convinced that the exponents mod 400 = $\phi(1000)$ and mod 100 are what count (i.e., don't be one of the many people who mistakenly try to work with the exponents mod 1000 and mod 101 in these examples).

3.6.1 Three-Pass Protocol

Alice wishes to transfer a secret key K (or any short message) to Bob via communication on a public channel. The Basic Principle can be used to solve this problem.

First, here is a nonmathematical way to do it. Alice puts K into a box and puts her lock on the box. She sends the locked box to Bob, who puts his lock on the box and sends the box back to Alice. Alice then takes her lock off and sends the box to Bob. Bob takes his lock off, opens the box, and finds K .

Here is the mathematical realization of the method.

First, Alice chooses a large prime number p that is large enough to represent the key K . For example, if Alice were trying to send a 56-bit key, she would need a prime number that is at least 56 bits long. However, for security purposes (to make what is known as the discrete log problem hard), she would want to choose a prime significantly longer than 56 bits. Alice publishes p so that Bob (or anyone else) can download it. Bob downloads p . Alice and Bob now do the following:

1. Alice selects a random number a with $\gcd(a, p - 1) = 1$ and Bob selects a random number b with $\gcd(b, p - 1) = 1$. We will denote by a^{-1} and b^{-1} the inverses of a and $b \pmod{p - 1}$.
2. Alice sends $K_1 \equiv K^a \pmod{p}$ to Bob.
3. Bob sends $K_2 \equiv K_1^b \pmod{p}$ to Alice.
4. Alice sends $K_3 \equiv K_2^{a^{-1}} \pmod{p}$ to Bob.
5. Bob computes $K \equiv K_3^{b^{-1}} \pmod{p}$.

At the end of this protocol, both Alice and Bob have the key K .

The reason this works is that Bob has computed $K^{aba^{-1}b^{-1}} \pmod{p}$. Since $aa^{-1} \equiv bb^{-1} \equiv 1 \pmod{p - 1}$, the Basic Principle implies that $K^{aba^{-1}b^{-1}} \equiv K^1 \equiv K \pmod{p}$.

The procedure is usually attributed to Shamir and to Massey and Omura. One drawback is that it requires multiple communications between Alice and Bob. Also, it

is vulnerable to the intruder-in-the-middle attack (see [Chapter 15](#)).

3.7 Primitive Roots

Consider the powers of 3 (mod 7):

$$3^1 \equiv 3, \quad 3^2 \equiv 2, \quad 3^3 \equiv 6, \quad 3^4 \equiv 4, \quad 3^5 \equiv 5, \quad 3^6 \equiv 1.$$

Note that we obtain all the nonzero congruence classes mod 7 as powers of 3. This means that 3 is a primitive root mod 7 (the term *multiplicative generator* might be better but is not as common). Similarly, every nonzero congruence class mod 13 is a power of 2, so 2 is a primitive root mod 13. However, $3^3 \equiv 1 \pmod{13}$, so the powers of 3 mod 13 repeat much more frequently:

$$3^1 \equiv 3, \quad 3^2 \equiv 9, \quad 3^3 \equiv 1, \quad 3^4 \equiv 3, \quad 3^5 \equiv 9, \quad 3^6 \equiv 1, \dots,$$

so only 1, 3, 9 are powers of 3. Therefore, 3 is not a primitive root mod 13. The primitive roots mod 13 are 2, 6, 7, 11.

In general, when p is a prime, a **primitive root** mod p is a number whose powers yield every nonzero class mod p . It can be shown that there are $\phi(p - 1)$ primitive roots mod p . In particular, there is always at least one. In practice, it is not difficult to find one, at least if the factorization of $p - 1$ is known. See [Exercise 54](#).

The following summarizes the main facts we need about primitive roots.

Proposition

Let α be a primitive root for the prime p .

1. Let n be an integer. Then $\alpha^n \equiv 1 \pmod{p}$ if and only if $n \equiv 0 \pmod{p - 1}$.

2. If j and k are integers, then $\alpha^j \equiv \alpha^k \pmod{p}$ if and only if $j \equiv k \pmod{p-1}$.
3. A number β is a primitive root mod p if and only if $p-1$ is the smallest positive integer k such that $\beta^k \equiv 1 \pmod{p}$.

Proof. If $n \equiv 0 \pmod{p-1}$, then $n = (p-1)m$ for some m . Therefore,

$$\alpha^n \equiv (\alpha^m)^{p-1} \equiv 1 \pmod{p}$$

by Fermat's theorem. Conversely, suppose $\alpha^n \equiv 1 \pmod{p}$. We want to show that $p-1$ divides n , so we divide $p-1$ into n and try to show that the remainder is 0. Write

$$n = (p-1)q + r, \quad \text{with } 0 \leq r < p-1$$

(this is just division with quotient q and remainder r).

We have

$$1 \equiv \alpha^n \equiv (\alpha^q)^{p-1} \alpha^r \equiv 1 \cdot \alpha^r \equiv \alpha^r \pmod{p}.$$

Suppose $r > 0$. If we consider the powers α, α^2, \dots of $\alpha \pmod{p}$, then we get back to 1 after r steps. Then

$$\alpha^{r+1} \equiv \alpha, \quad \alpha^{r+2} \equiv \alpha^2, \quad \dots$$

so the powers of $\alpha \pmod{p}$ yield only the r numbers $\alpha, \alpha^2, \dots, 1$. Since $r < p-1$, not every number mod p can be a power of α . This contradicts the assumption that α is a primitive root.

The only possibility that remains is that $r = 0$. This means that $n = (p-1)q$, so $p-1$ divides n . This proves part (1).

For part (2), assume that $j \geq k$ (if not, switch j and k). Suppose that $\alpha^j \equiv \alpha^k \pmod{p}$. Dividing both sides by α^k yields $\alpha^{j-k} \equiv 1 \pmod{p}$. By part (1), $j-k \equiv 0 \pmod{p-1}$, so $j \equiv k \pmod{p-1}$. Conversely, if $j \equiv k \pmod{p-1}$, then $j-k \equiv 0 \pmod{p-1}$, so $\alpha^{j-k} \equiv 1 \pmod{p}$, again by part (1). Multiplying by α^k yields the result.

For part (3), if β is a primitive root, then part (1) says that any integer k with $\beta^k \equiv 1 \pmod{p}$ must be a multiple of $p - 1$, so $k = p - 1$ is the smallest.

Conversely, suppose $k = p - 1$ is the smallest. Look at the numbers $1, \beta, \beta^2, \dots, \beta^{p-2} \pmod{p}$. If two are congruent mod p , say $\beta^i \equiv \beta^j$ with

$0 \leq i < j \leq p - 2$, then $\beta^{j-i} \equiv 1 \pmod{p}$ (note: $\beta^{p-1} \equiv 1 \pmod{p}$ implies that $\beta \not\equiv 0 \pmod{p}$, so we can divide by β). Since $0 < j - i < p - 1$, this contradicts the assumption that $k = p - 1$ is smallest.

Therefore, the numbers $1, \beta, \beta^2, \dots, \beta^{p-2}$ must be distinct mod p . Since there are $p - 1$ numbers on this list and there are $p - 1$ numbers $1, 2, 3, \dots, p - 1 \pmod{p}$, the two lists must be the same, up to order.

Therefore, each number on the list $1, 2, 3, \dots, p - 1$ is congruent to a power of β , so β is a primitive root mod p .

Warning: α is a primitive root mod p if and only if $p - 1$ is the smallest positive n such that $\alpha^n \equiv 1 \pmod{p}$. If you want to prove that α is a primitive root, it does not suffice to prove that $\alpha^{p-1} \equiv 1 \pmod{p}$. After all, Fermat's theorem says that every α satisfies this, as long as $\alpha \not\equiv 0 \pmod{p}$. To prove that α is a primitive root, you must show that $p - 1$ is the *smallest* positive exponent k such that $\alpha^k \equiv 1$.

3.8 Inverting Matrices Mod n

Finding the inverse of a matrix mod n can be accomplished by the usual methods for inverting a matrix, as long as we apply the rule given in [Section 3.3](#) for dealing with fractions. The basic fact we need is that a square matrix is invertible mod n if and only if its determinant and n are relatively prime.

We treat only small matrices here, since that is all we need for the examples in this book. In this case, the easiest way is to find the inverse of the matrix is to use rational numbers, then change back to numbers mod n . It is a general fact that the inverse of an integer matrix can always be written as another integer matrix divided by the determinant of the original matrix. Since we are assuming the determinant and n are relatively prime, we can invert the determinant as in [Section 3.3](#).

For example, in the 2×2 case the usual formula is

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix},$$

so we need to find an inverse for $ad - bc \pmod{n}$.

Example

Suppose we want to invert $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \pmod{11}$. Since $ad - bc = -2$, we need the inverse of $-2 \pmod{11}$. Since $5 \times (-2) \equiv 1 \pmod{11}$, we can replace $-1/2$ by 5 and obtain

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}^{-1} \equiv \frac{-1}{2} \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} \equiv 5 \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} \equiv \begin{pmatrix} 9 & 1 \\ 7 & 5 \end{pmatrix} \pmod{11}.$$

A quick calculation shows that

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 9 & 1 \\ 7 & 5 \end{pmatrix} = \begin{pmatrix} 23 & 11 \\ 55 & 23 \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \pmod{11}.$$

Example

Suppose we want the inverse of

$$M = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \end{pmatrix} \pmod{11}.$$

The determinant is 2 and the inverse of M in rational numbers is

$$\frac{1}{2} \begin{pmatrix} 6 & -5 & 1 \\ -6 & 8 & -2 \\ 2 & -3 & 1 \end{pmatrix}.$$

(For ways to calculate the inverse of a matrix, look at any book on linear algebra.) We can replace $1/2$ with 6 mod 11 and obtain

$$M^{-1} \equiv \begin{pmatrix} 3 & 3 & 6 \\ 8 & 4 & 10 \\ 1 & 4 & 6 \end{pmatrix} \pmod{11}.$$

Why do we need the determinant and n to be relatively prime? Suppose $MN \equiv I \pmod{n}$, where I is the identity matrix. Then

$$\det(M) \det(N) \equiv \det(MN) \equiv \det(I) \equiv 1 \pmod{n}.$$

Therefore, $\det(M)$ has an inverse mod n , which means that $\det(M)$ and n must be relatively prime.

3.9 Square Roots Mod n

Suppose we are told that $x^2 \equiv 71 \pmod{77}$ has a solution. How do we find one solution, and how do we find all solutions? More generally, consider the problem of finding all solutions of $x^2 \equiv b \pmod{n}$, where $n = pq$ is the product of two primes. We show in the following that this can be done quite easily, once the factorization of n is known. Conversely, if we know all solutions, then it is easy to factor n .

Let's start with the case of square roots mod a prime p . The easiest case is when $p \equiv 3 \pmod{4}$, and this suffices for our purposes. The case when $p \equiv 1 \pmod{4}$ is more difficult. See [Cohen, pp. 31–34] or [KraftW, p. 317].

Proposition

Let $p \equiv 3 \pmod{4}$ be prime and let y be an integer. Let $x \equiv y^{(p+1)/4} \pmod{p}$.

1. If y has a square root mod p , then the square roots of $y \pmod{p}$ are $\pm x$.
2. If y has no square root mod p , then $-y$ has a square root mod p , and the square roots of $-y$ are $\pm x$.

Proof. If $y \equiv 0 \pmod{p}$, all the statements are trivial, so assume $y \not\equiv 0 \pmod{p}$. Fermat's theorem says that $y^{p-1} \equiv 1 \pmod{p}$. Therefore,

$$x^4 \equiv y^{p+1} \equiv y^2 y^{p-1} \equiv y^2 \pmod{p}.$$

This implies that $(x^2 + y)(x^2 - y) \equiv 0 \pmod{p}$, so $x^2 \equiv \pm y \pmod{p}$. (See Exercise 13(a).) Therefore, at least one of y and $-y$ is a square mod p . Suppose both y

and $-y$ are squares mod p , say $y \equiv a^2$ and $-y \equiv b^2$. Then $-1 \equiv (a/b)^2$ (work with fractions mod p as in Section 3.3), which means -1 is a square mod p . This is impossible when $p \equiv 3 \pmod{4}$ (see Exercise 26). Therefore, exactly one of y and $-y$ has a square root mod p . If y has a square root mod p then $y \equiv x^2$, and the two square roots of y are $\pm x$. If $-y$ has a square root, then $x^2 \equiv -y$.

Example

Let's find the square root of 5 mod 11. Since $(p+1)/4 = 3$, we compute $x \equiv 5^3 \equiv 4 \pmod{11}$. Since $4^2 \equiv 5 \pmod{11}$, the square roots of 5 mod 11 are ± 4 .

Now let's try to find a square root of 2 mod 11. Since $(p+1)/4 = 3$, we compute $2^3 \equiv 8 \pmod{11}$. But $8^2 \equiv 9 \equiv -2 \pmod{11}$, so we have found a square root of -2 rather than of 2. This is because 2 has no square root mod 11.

We now consider square roots for a composite modulus. Note that

$$x^2 \equiv 71 \pmod{77}$$

means that

$$x^2 \equiv 71 \equiv 1 \pmod{7} \text{ and } x^2 \equiv 71 \equiv 5 \pmod{11}.$$

Therefore,

$$x \equiv \pm 1 \pmod{7} \text{ and } x \equiv \pm 4 \pmod{11}.$$

The Chinese remainder theorem tells us that a congruence mod 7 and a congruence mod 11 can be recombined into a congruence mod 77. For example, if $x \equiv 1 \pmod{7}$ and $x \equiv 4 \pmod{11}$, then

$x \equiv 15 \pmod{77}$. In this way, we can recombine in four ways to get the solutions

$$x \equiv \pm 15, \pm 29 \pmod{77}.$$

Now let's turn things around. Suppose $n = pq$ is the product of two primes and we know the four solutions $x \equiv \pm a, \pm b$ of $x^2 \equiv y \pmod{n}$. From the construction just used above, we know that

$a \equiv b \pmod{p}$ and $a \equiv -b \pmod{q}$ (or the same congruences with p and q switched). Therefore, $p|(a - b)$ but $q \nmid (a - b)$. This means that $\gcd(a - b, n) = p$, so we have found a nontrivial factor of n (this is essentially the Basic Factorization Principle of [Section 9.4](#)).

For example, in the preceding example we know that $15^2 \equiv 29^2 \equiv 71 \pmod{77}$. Therefore, $\gcd(15 - 29, 77) = 7$ gives a nontrivial factor of 77.

Another example of computing square roots mod n is given in [Section 18.1](#).

Notice that all the operations used above are fast, with the exception of factoring n . In particular, the Chinese remainder theorem calculation can be done quickly. So can the computation of the gcd. The modular exponentiations needed to compute square roots mod p and mod q can be done quickly using successive squaring. Therefore, we can state the following principle:

Suppose $n = pq$ is the product of two primes congruent to 3 mod 4, and suppose y is a number relatively prime to n that has a square root mod n . Then finding the four solutions $x \equiv \pm a, \pm b$ to $x^2 \equiv y \pmod{n}$ is computationally equivalent to factoring n .

In other words, if we can find the solutions, then we can easily factor n ; conversely, if we can factor n , we can

easily find the solutions. For more on this, see [Section 9.4](#).

Now suppose someone has a machine that can find single square roots mod n . That is, if we give the machine a number y that has a square root mod n , then the machine returns one solution of $x^2 \equiv y \pmod{n}$. We can use this machine to factor n as follows: Choose a random integer $x_1 \pmod{n}$, compute $y \equiv x_1^2 \pmod{n}$, and give the machine y . The machine returns x with $x^2 \equiv y \pmod{n}$. If our choice of x_1 is truly random, then the machine has no way of knowing the value of x_1 , hence it does not know whether $x \equiv x_1 \pmod{n}$ or not, even if it knows all four square roots of y . So half of the time, $x \equiv \pm x_1 \pmod{n}$, but half of the time, $x \not\equiv \pm x_1 \pmod{n}$. In the latter case, we compute $\gcd(x - x_1, n)$ and obtain a nontrivial factor of n . Since there is a 50% chance of success for each time we choose x_1 , if we choose several random values of x_1 , then it is very likely that we will eventually factor n . Therefore, we conclude that any machine that can find single square roots mod n can be used, with high probability, to factor n .

3.10 Legendre and Jacobi Symbols

Suppose we want to determine whether or not $x^2 \equiv a \pmod{p}$ has a solution, where p is prime. If p is small, we could square all of the numbers mod p and see if a is on the list. When p is large, this is impractical. If $p \equiv 3 \pmod{4}$, we can use the technique of the previous section and compute $s \equiv a^{(p+1)/4} \pmod{p}$. If a has a square root, then s is one of them, so we simply have to square s and see if we get a . If not, then a has no square root mod p . The following proposition gives a method for deciding whether a is a square mod p that works for arbitrary odd p .

Proposition

Let p be an odd prime and let a be an integer with $a \not\equiv 0 \pmod{p}$. Then $a^{(p-1)/2} \equiv \pm 1 \pmod{p}$. The congruence $x^2 \equiv a \pmod{p}$ has a solution if and only if $a^{(p-1)/2} \equiv 1 \pmod{p}$.

Proof. Let $y \equiv a^{(p-1)/2} \pmod{p}$. Then $y^2 \equiv a^{p-1} \equiv 1 \pmod{p}$, by Fermat's theorem. Therefore (Exercise 15), $y \equiv \pm 1 \pmod{p}$.

If $a \equiv x^2$, then $a^{(p-1)/2} \equiv x^{p-1} \equiv 1 \pmod{p}$. The hard part is showing the converse. Let α be a primitive root mod p . Then $a \equiv \alpha^j$ for some j . If $a^{(p-1)/2} \equiv 1 \pmod{p}$, then

$$\alpha^{j(p-1)/2} \equiv a^{(p-1)/2} \equiv 1 \pmod{p}.$$

By the Proposition of Section 3.7, $j(p-1)/2 \equiv 0 \pmod{p-1}$. This implies that j must

be even: $j = 2k$. Therefore, $a \equiv g^j \equiv (\alpha^k)^2 \pmod{p}$, so a is a square mod p .

The criterion is very easy to implement on a computer, but it can be rather difficult to use by hand. In the following, we introduce the Legendre and Jacobi symbols, which give us an easy way to determine whether or not a number is a square mod p . They also are useful in primality testing (see Section 9.3).

Let p be an odd prime and let $a \not\equiv 0 \pmod{p}$. Define the **Legendre symbol**

$$\left(\frac{a}{p}\right) = \begin{cases} +1 & \text{if } x^2 \equiv a \pmod{p} \text{ has a solution.} \\ -1 & \text{if } x^2 \equiv a \pmod{p} \text{ has no solution.} \end{cases}$$

Some important properties of the Legendre symbol are given in the following.

Proposition

Let p be an odd prime.

1. If $a \equiv b \pmod{p}$, then

$$\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right).$$

2. If $a \not\equiv 0 \pmod{p}$, then

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

3. If $ab \not\equiv 0 \pmod{p}$, then

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right) \left(\frac{b}{p}\right).$$

$$4. \quad \left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}.$$

Proof. Part (1) is true because the solutions to $X^2 \equiv a \pmod{p}$ are the same as those to $X^2 \equiv b \pmod{p}$ when $a \equiv b \pmod{p}$.

Part (2) is the definition of the Legendre symbol combined with the previous proposition.

To prove part (3), we use part (2):

$$\left(\frac{ab}{p}\right) \equiv (ab)^{(p-1)/2} \equiv a^{(p-1)/2}b^{(p-1)/2} \equiv \left(\frac{a}{p}\right)\left(\frac{b}{p}\right) \pmod{p}.$$

Since the left and right ends of this congruence are ± 1 and they are congruent mod the odd prime p , they must be equal. This proves (3).

For part (4), use part (2) with $a = -1$:

$$\left(\frac{-1}{p}\right) \equiv (-1)^{(p-1)/2} \pmod{p}.$$

Again, since the left and right sides of this congruence are ± 1 and they are congruent mod the odd prime p , they must be equal. This proves (4).

Example

Let $p = 11$. The nonzero squares mod 11 are 1, 3, 4, 5, 9. We have

$$\left(\frac{6}{11}\right)\left(\frac{7}{11}\right) = (-1)(-1) = +1$$

and (use property (1))

$$\left(\frac{42}{11}\right) = \left(\frac{9}{11}\right) = +1.$$

Therefore,

$$\left(\frac{6}{11}\right)\left(\frac{7}{11}\right) = \left(\frac{42}{11}\right).$$

The Jacobi symbol extends the Legendre symbol from primes p to composite odd integers n . One might be tempted to define the symbol to be $+1$ if a is a square mod n and -1 if not. However, this would cause the

important property (3) to fail. For example, 2 is not a square mod 35, and 3 is not a square mod 35 (since they are not squares mod 5), but also the product 6 is not a square mod 35 (since it is not a square mod 7). If Property (3) held, then we would have $(-1)(-1) = -1$, which is false.

In order to preserve property (3), we define the **Jacobi symbol** as follows. Let n be an odd positive integer and let a be a nonzero integer with $\gcd(a, n) = 1$. Let

$$n = p_1^{b_1} p_2^{b_2} \cdots p_r^{b_r}$$

be the prime factorization of n . Then

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{b_1} \left(\frac{a}{p_2}\right)^{b_2} \cdots \left(\frac{a}{p_r}\right)^{b_r}.$$

The symbols on the right side are the Legendre symbols introduced earlier. Note that if $n = p$, the right side is simply one Legendre symbol, so the Jacobi symbol reduces to the Legendre symbol.

Example

Let $n = 135 = 3^3 \cdot 5$. Then

$$\left(\frac{2}{135}\right) = \left(\frac{2}{3}\right)^3 \left(\frac{2}{5}\right) = (-1)^3(-1) = +1.$$

Note that 2 is not a square mod 5, hence is not a square mod 135. Therefore, the fact that the Jacobi symbol has the value +1 does not imply that 2 is a square mod 135.

The main properties of the Jacobi symbol are given in the following theorem. Parts (1), (2), and (3) can be deduced from those of the Legendre symbol. Parts (4) and (5) are much deeper.

Theorem

Let n be odd.

1. If $a \equiv b \pmod{n}$ and $\gcd(a, n) = 1$, then

$$\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right).$$

2. If $\gcd(ab, n) = 1$, then

$$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right) \left(\frac{b}{n}\right).$$

3. $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}.$

4. $\left(\frac{2}{n}\right) = \begin{cases} +1 & \text{if } n \equiv 1 \text{ or } 7 \pmod{8} \\ -1 & \text{if } n \equiv 3 \text{ or } 5 \pmod{8}. \end{cases}$

5. Let m be odd with $\gcd(m, n) = 1$. Then

$$\left(\frac{m}{n}\right) = \begin{aligned} &- \left(\frac{n}{m}\right) \text{ if } m \equiv n \equiv 3 \pmod{4} \\ &+ \left(\frac{n}{m}\right) \text{ otherwise.} \end{aligned}$$

Note that we did not include a statement that

$\left(\frac{a}{n}\right) \equiv a^{(n-1)/2}$. This is usually not true for composite n (see [Exercise 45](#)). In fact, the Solovay-Strassen primality test (see [Section 9.3](#)) is based on this fact.

Part (5) is the famous **law of quadratic reciprocity**, proved by Gauss in 1796. When m and n are primes, it relates the question of whether m is a square mod n to the question of whether n is a square mod m .

A proof of the theorem when m and n are primes can be found in most elementary number theory texts. The extension to composite m and n can be deduced fairly easily from this case. See [Niven et al.], [Rosen], or [KraftW], for example.

When quadratic reciprocity is combined with the other properties of the Jacobi symbol, we obtain a fast way to

evaluate the symbol. Here are two examples.

Example

Let's calculate $\left(\frac{4567}{12345}\right)$:

$$\begin{aligned}
 \left(\frac{4567}{12345}\right) &= +\left(\frac{12345}{4567}\right) \quad (\text{by (5), since } 12345 \equiv 1 \pmod{4}) \\
 &= +\left(\frac{3211}{4567}\right) \quad (\text{by (1), since } 12345 \equiv 3211 \pmod{4567}) \\
 &= -\left(\frac{4567}{3211}\right) \quad (\text{by (5)}) = -\left(\frac{1356}{3211}\right) \quad (\text{by (1)}) \\
 &= -\left(\frac{2}{3211}\right)^2 \left(\frac{339}{3211}\right) \quad (\text{by (2), since } 1356 = 2^2 \cdot 339) \\
 &= -\left(\frac{339}{3211}\right) \quad (\text{since } (\pm 1)^2 = 1) \\
 &= +\left(\frac{3211}{339}\right) \quad (\text{by (5)}) = +\left(\frac{160}{339}\right) \quad (\text{by (1)}) \\
 &= +\left(\frac{2}{339}\right)^5 \left(\frac{5}{339}\right) \quad (\text{by (2), since } 160 = 2^5 \cdot 5) \\
 &= +(-1)^5 \left(\frac{5}{339}\right) \quad (\text{by (4)}) = -\left(\frac{339}{5}\right) \quad (\text{by (5)}) \\
 &= -\left(\frac{4}{5}\right) \quad (\text{by (1)}) = -\left(\frac{2}{5}\right)^2 = -1.
 \end{aligned}$$

The only factorization needed in the calculation was removing powers of 2, which is easy to do. The fact that the calculations can be done without factoring odd numbers is important in the applications. The fact that the answer is -1 implies that 4567 is not a square mod 12345. However, if the answer had been $+1$, we could not have deduced whether 4567 is a square or is not a square mod 12345. See [Exercise 44](#).

Example

Let's calculate $\left(\frac{107}{137}\right)$:

$$\begin{aligned}
\left(\frac{107}{137}\right) &= +\left(\frac{137}{107}\right) \quad (\text{by (5)}) \\
&= +\left(\frac{30}{107}\right) \quad (\text{by (1)}) \\
&= +\left(\frac{2}{107}\right)\left(\frac{15}{107}\right) \quad (\text{by (2)}) \\
&= +(-1)\left(\frac{15}{107}\right) \quad (\text{by (4)}) \\
&= +\left(\frac{107}{15}\right) \quad (\text{by (5)}) \\
&= +\left(\frac{2}{15}\right) \quad (\text{by (1)}) \\
&= +1 \quad (\text{by (5)}).
\end{aligned}$$

Since 137 is a prime, this says that 107 is a square mod 137. In contrast, during the calculation, we used the fact that $\left(\frac{2}{15}\right) = +1$. This does not mean that 2 is a square mod 15. In fact, 2 is not a square mod 5, so it cannot be a square mod 15. Therefore, although we can interpret the final answer as saying that 107 is a square mod the prime 137, we should not interpret intermediate steps involving composite numbers as saying that a number is a square.

Suppose $n = pq$ is the product of two large primes. If $\left(\frac{a}{n}\right) = -1$, then we can conclude that a is not a square mod n . What can we conclude if $\left(\frac{a}{n}\right) = +1$? Since

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right),$$

there are two possibilities:

$$\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = -1 \quad \text{or} \quad \left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = +1.$$

In the first case, a is not a square mod p , therefore cannot be a square mod pq .

In the second case, a is a square mod p and mod q . The Chinese remainder theorem can be used to combine a square root mod p and a square root mod q to get a square root of a mod n . Therefore, a is a square mod n .

Therefore, if $\left(\frac{a}{n}\right) = +1$, then a can be either a square or a nonsquare mod n . Deciding which case holds is called the **quadratic residuosity problem**. No fast algorithm is known for solving it. Of course, if we can factor n , then the problem can easily be solved by computing $\left(\frac{a}{p}\right)$.

3.11 Finite Fields

Note: This section is more advanced than the rest of the chapter. It is included because finite fields are often used in cryptography. In particular, finite fields appear in four places in this book. The finite field $GF(2^8)$ is used in AES (Chapter 8). Finite fields give an explanation of some phenomena that are mentioned in Section 5.2. Finally, finite fields are used in Section 21.4, Chapter 22 and in error correcting codes (Chapter 24).

Many times throughout this book, we work with the integers mod p , where p is a prime. We can add, subtract, and multiply, but what distinguishes working mod p from working mod an arbitrary integer n is that we can divide by any number that is nonzero mod p . For example, if we need to solve $3x \equiv 1 \pmod{5}$, then we divide by 3 to obtain $x \equiv 2 \pmod{5}$. In contrast, if we want to solve $3x \equiv 1 \pmod{6}$, there is no solution since we cannot divide by 3 mod 6. Loosely speaking, a set that has the operations of addition, multiplication, subtraction, and division by nonzero elements is called a field. We also require that the associative, commutative, and distributive laws hold.

Example

The basic examples of fields are the real numbers, the complex numbers, the rational numbers, and the integers mod a prime. The set of all integers is not a field since we sometimes cannot divide and obtain an answer in the set (for example, $4/3$ is not an integer).

Example

Here is a field with four elements. Consider the set

$$GF(4) = \{0, 1, \omega, \omega^2\},$$

with the following laws:

1. $0 + x = x$ for all x .
2. $x + x = 0$ for all x .
3. $1 \cdot x = x$ for all x .
4. $\omega + 1 = \omega^2$.
5. Addition and multiplication are commutative and associative, and the distributive law $x(y + z) = xy + xz$ holds for all x, y, z .

Since

$$\omega^3 = \omega \cdot \omega^2 = \omega \cdot (1 + \omega) = \omega + \omega^2 = \omega + (1 + \omega) = 1,$$

we see that ω^2 is the multiplicative inverse of ω .

Therefore, every nonzero element of $GF(4)$ has a multiplicative inverse, and $GF(4)$ is a field with four elements.

In general, a **field** is a set containing elements **0** and **1** (with $1 \neq 0$) and satisfying the following:

1. It has a multiplication and addition satisfying (1), (3), (5) in the preceding list.
2. Every element has an additive inverse (for each x , this means there exists an element $-x$ such that $x + (-x) = 0$).
3. Every nonzero element has a multiplicative inverse.

A field is closed under subtraction. To compute $x - y$, simply compute $x + (-y)$.

The set of 2×2 matrices with real entries is not a field for two reasons. First, the multiplication is not commutative. Second, there are nonzero matrices that do not have inverses (and therefore we cannot divide by them). The set of nonnegative real numbers is not a field. We can add, multiply, and divide, but sometimes when we subtract the answer is not in the set.

For every power p^n of a prime, there is exactly one finite field with p^n elements, and these are the only finite fields. We'll soon show how to construct them, but first let's point out that if $n > 1$, then the integers mod p^n do not form a field. The congruence $px \equiv 1 \pmod{p^n}$ does not have a solution, so we cannot divide by p , even though $p \not\equiv 0 \pmod{p^n}$. Therefore, we need more complicated constructions to produce fields with p^n elements.

The field with p^n elements is called $GF(p^n)$. The “GF” is for “Galois field,” named for the French mathematician Evariste Galois (1811–1832), who did some early work related to fields.

Example, continued

Here is another way to produce the field $GF(4)$. Let $\mathbf{Z}_2[X]$ be the set of polynomials whose coefficients are integers mod 2. For example, $1 + X^3 + X^6$ and X are in this set. Also, the constant polynomials 0 and 1 are in $\mathbf{Z}_2[X]$. We can add, subtract, and multiply in this set, as long as we work with the coefficients mod 2. For example,

$$(X^3 + X + 1)(X + 1) = X^4 + X^3 + X^2 + 1$$

since the term $2X$ disappears mod 2. The important property for our purposes is that we can perform division with remainder, just as with the integers. For example, suppose we divide $X^2 + X + 1$ into $X^4 + X^3 + 1$. We can do this by long division, just as with numbers:

$$\begin{array}{r} X^2 + 1 \\ X^2 + X + 1 \end{array} \overline{\Big|} \begin{array}{r} X^4 + X^3 + 1 \\ X^4 + X^3 + X^2 \\ X^2 + 1 \\ X^2 + X + 1 \\ X \end{array}$$

In words, what we did was to divide by $X^2 + X + 1$ and obtain the X^2 as the first term of the quotient. Then we multiplied this X^2 times $X^2 + X + 1$ to get $X^4 + X^3 + X^2$, which we subtracted from $X^4 + X^3 + 1$, leaving $X^2 + 1$. We divided this $X^2 + 1$ by $X^2 + X + 1$ and obtained the second term of the quotient, namely 1. Multiplying 1 times $X^2 + X + 1$ and subtracting from $X^2 + 1$ left the remainder X . Since the degree of the polynomial X is less than the degree of $X^2 + X + 1$, we stopped. The quotient was $X^2 + 1$ and the remainder was X :

$$X^4 + X^3 + 1 = (X^2 + 1)(X^2 + X + 1) + X.$$

We can write this as

$$X^4 + X^3 + 1 \equiv X \pmod{X^2 + X + 1}.$$

Whenever we divide by $X^2 + X + 1$ we can obtain a remainder that is either 0 or a polynomial of degree at most 1 (if the remainder had degree 2 or more, we could continue dividing). Therefore, we define $\mathbf{Z}_2[X] \pmod{X^2 + X + 1}$ to be the set

$$\{0, 1, X, X + 1\}$$

of polynomials of degree at most 1, since these are the remainders that we obtain when we divide by $X^2 + X + 1$. Addition, subtraction, and multiplication are done mod $X^2 + X + 1$. This is completely analogous to what happens when we work with integers mod n . In the present situation, we say that two polynomials $f(X)$ and $g(X)$ are congruent mod $X^2 + X + 1$, written $f(X) \equiv g(X) \pmod{X^2 + X + 1}$, if $f(X)$ and $g(X)$ have the same remainder when divided by $X^2 + X + 1$. Another way of saying this is that $f(X) - g(X)$ is a multiple of $X^2 + X + 1$. This means that there is a polynomial $h(X)$ such that $f(X) - g(X) = (X^2 + X + 1)h(X)$.

Now let's multiply in $\mathbf{Z}_2[X] \pmod{X^2 + X + 1}$. For example,

$$X \cdot X = X^2 \equiv X + 1 \pmod{X^2 + X + 1}.$$

(It might seem that the right side should be $-X - 1$, but recall that we are working with coefficients mod 2, so $+1$ and -1 are the same.) As another example, we have

$$X^3 \equiv X \cdot X^2 \equiv X \cdot (X + 1) \equiv X^2 + X \equiv 1 \pmod{X^2 + X + 1}.$$

It is easy to see that we are working with the set $GF(4)$ from before, with X in place of ω .

Working with $\mathbf{Z}_2[X]$ mod a polynomial can be used to produce finite fields. But we cannot work mod an arbitrary polynomial. The polynomial must be irreducible, which means that it doesn't factor into polynomials of lower degree mod 2. For example, $X^2 + 1$, which is irreducible when we are working with real numbers, is not irreducible when the coefficients are taken mod 2 since $X^2 + 1 = (X + 1)(X + 1)$ when we are working mod 2. However, $X^2 + X + 1$ is irreducible: Suppose it factors mod 2 into polynomials of lower degree. The only possible factors mod 2 are X and $X + 1$, and $X^2 + X + 1$ is not a multiple of either of these, even mod 2.

Here is the general procedure for constructing a finite field with p^n elements, where p is prime and $n \geq 1$. We let \mathbf{Z}_p denote the integers mod p .

1. $\mathbf{Z}_p[X]$ is the set of polynomials with coefficients mod p .
2. Choose $P(X)$ to be an irreducible polynomial mod p of degree n .
3. Let $GF(p^n)$ be $\mathbf{Z}_p[X] \pmod{P(X)}$. Then $GF(p^n)$ is a field with p^n elements.

The fact that $GF(p^n)$ has p^n elements is easy to see. The possible remainders after dividing by $P(X)$ are the polynomials of the form $a_0 + a_1X + \cdots + a_{n-1}X^{n-1}$, where the coefficients

are integers mod p . There are p choices for each coefficient, hence p^n possible remainders.

For each n , there are irreducible polynomials mod p of degree n , so this construction produces fields with p^n elements for each $n \geq 1$. What happens if we do the same construction for two different polynomials $P_1(X)$ and $P_2(X)$, both of degree n ? We obtain two fields, call them $GF(p^n)'$ and $GF(p^n)''$. It is possible to show that these are essentially the same field (the technical term is that the two fields are isomorphic), though this is not obvious since multiplication mod $P_1(X)$ is not the same as multiplication mod $P_2(X)$.

3.11.1 Division

We can easily add, subtract, and multiply polynomials in $\mathbf{Z}_p[X]$, but division is a little more subtle. Let's look at an example. The polynomial $X^8 + X^4 + X^3 + X + 1$ is irreducible in $\mathbf{Z}_2[X]$ (although there are faster methods, one way to show it is irreducible is to divide it by all polynomials of smaller degree in $\mathbf{Z}_2[X]$). Consider the field

$$GF(2^8) = \mathbf{Z}_2[X] \pmod{X^8 + X^4 + X^3 + X + 1}.$$

Since $X^7 + X^6 + X^3 + X + 1$ is not 0, it should have an inverse. The inverse is found using the analog of the extended Euclidean algorithm. First, perform the gcd calculation for

$$\gcd(X^7 + X^6 + X^3 + X + 1, X^8 + X^4 + X^3 + X + 1).$$

The procedure (remainder \rightarrow divisor \rightarrow dividend \rightarrow ignore) is the same as for integers:

$$\begin{aligned} X^8 + X^4 + X^3 + X + 1 &= (X+1)(X^7 + X^6 + X^3 + X + 1) + (X^6 + X^2 + X) \\ X^7 + X^6 + X^3 + X + 1 &= (X+1)(X^6 + X^2 + X) + 1. \end{aligned}$$

The last remainder is 1, which tells us that the “greatest common divisor” of $X^7 + X^6 + X^3 + X + 1$ and $X^8 + X^4 + X^3 + X + 1$ is 1. Of course, this must be

the case, since $X^8 + X^4 + X^3 + X + 1$ is irreducible, so its only factors are 1 and itself.

Now work the Extended Euclidean algorithm to express 1 as a linear combination of $X^7 + X^6 + X^3 + X + 1$ and $X^8 + X^4 + X^3 + X + 1$:

x	y
$X^8 + X^4 + X^3 + X + 1$	1
$X^7 + X^6 + X^3 + X + 1$	0
$X^6 + X^2 + X$	1
	$X + 1$
	(1st row) $-(X + 1) \cdot$ (2nd row)
1	$X + 1$
	X^2
	(2nd row) $-(X + 1) \cdot$ $(3\text{rd row}).$

The end result is

$$1 = (X^2)(X^7 + X^6 + X^3 + X + 1) + (X + 1)(X^8 + X^4 + X^3 + X + 1).$$

Reducing mod $X^8 + X^4 + X^3 + X + 1$, we obtain

$$(X^2)(X^7 + X^6 + X^3 + X + 1) \equiv 1 \pmod{X^8 + X^4 + X^3 + X + 1},$$

which means that X^2 is the multiplicative inverse of $X^7 + X^6 + X^3 + X + 1$. Whenever we need to divide by $X^7 + X^6 + X^3 + X + 1$, we can instead multiply by X^2 . This is the analog of what we did when working with the usual integers mod p .

3.11.2 $GF(2^8)$

In Chapter 8, we discuss AES, which uses $GF(2^8)$, so let's look at this field a little more closely. We'll work mod the irreducible polynomial

$X^8 + X^4 + X^3 + X + 1$, since that is the one used by AES. However, there are other irreducible polynomials of degree 8, and any one of them would lead to similar calculations. Every element can be represented uniquely as a polynomial

$$b_7X^7 + b_6X^6 + b_5X^5 + b_4X^4 + b_3X^3 + b_2X^2 + b_1X + b_0,$$

where each b_i is 0 or 1. The 8 bits $b_7b_6b_5b_4b_3b_2b_1b_0$ represent a byte, so we can represent the elements of $GF(2^8)$ as 8-bit bytes. For example, the polynomial $X^7 + X^6 + X^3 + X + 1$ becomes 11001011.

Addition is the XOR of the bits:

$$\begin{aligned} & (X^7 + X^6 + X^3 + X + 1) + (X^4 + X^3 + 1) \\ & \rightarrow 11001011 \oplus 00011001 = 11010010 \\ & \rightarrow X^7 + X^6 + X^4 + X. \end{aligned}$$

Multiplication is more subtle and does not have as easy an interpretation. That is because we are working mod the polynomial $X^8 + X^4 + X^3 + X + 1$, which we can represent by the 9 bits 100011011. First, let's multiply $X^7 + X^6 + X^3 + X + 1$ by X : With polynomials, we calculate

$$\begin{aligned} & (X^7 + X^6 + X^3 + X + 1)(X) = X^8 + X^7 + X^4 + X^2 + X \\ & = (X^7 + X^3 + X^2 + 1) + (X^8 + X^4 + X^3 + X + 1) \\ & \equiv X^7 + X^3 + X^2 + 1 \pmod{X^8 + X^4 + X^3 + X + 1}. \end{aligned}$$

The same operation with bits becomes

$$\begin{aligned} 11001011 & \rightarrow 110010110 \quad (\text{shift left and append a 0}) \\ & \rightarrow 110010110 \oplus 100011011 \quad (\text{subtract } X^8 + X^4 + X^3 + X + 1) \\ & = 010001101, \end{aligned}$$

which corresponds to the preceding answer. In general, we can multiply by X by the following algorithm:

1. Shift left and append a 0 as the last bit.
2. If the first bit is 0, stop.
3. If the first bit is 1, XOR with 100011011.

The reason we stop in step 2 is that if the first bit is 0 then the polynomial still has degree less than 8 after we multiply by X , so it does not need to be reduced. To multiply by higher powers of X , multiply by X several times. For example, multiplication by X^3 can be done with three shifts and at most three XORs. Multiplication by an arbitrary polynomial can be accomplished by multiplying by the various powers of X appearing in that polynomial, then adding (i.e., XORing) the results.

In summary, we see that the field operations of addition and multiplication in $GF(2^8)$ can be carried out very efficiently. Similar considerations apply to any finite field.

The analogy between the integers mod a prime and polynomials mod an irreducible polynomial is quite remarkable. We summarize in the following.

$$\begin{aligned} \text{integers} &\longleftrightarrow \mathbf{Z}_p[X] \\ \text{prime number } q &\longleftrightarrow \text{irreducible } P(X) \text{ of degree } n \\ \mathbf{Z}_q &\longleftrightarrow \mathbf{Z}_p[X] \pmod{P(X)} \\ \text{field with } q \text{ elements} &\longleftrightarrow \text{field with } p^n \text{ elements} \end{aligned}$$

Let $GF(p^n)^*$ denote the nonzero elements of $GF(p^n)$. This set, which has $p^n - 1$ elements, is closed under multiplication, just as the integers not congruent to 0 mod p are closed under multiplication. It can be shown that there is a generating polynomial $g(X)$ such that every element in $GF(p^n)^*$ can be expressed as a power of $g(X)$. This also means that the smallest exponent k such that $g(X)^k \equiv 1$ is $p^n - 1$. This is the analog of a primitive root for primes. There are $\phi(p^n - 1)$ such generating polynomials, where ϕ is Euler's function. An interesting situation occurs when $p = 2$ and $2^n - 1$ is prime. In this case, every nonzero polynomial $f(X) \neq 1$ in $GF(2^n)$ is a generating polynomial. (*Remark, for those who know some group theory:* The set $GF(2^n)^*$ is a group of prime order in this case, so every element except the identity is a generator.)

The **discrete log problem** mod a prime, which we'll discuss in Chapter 10, has an analog for finite fields; namely, given $h(x)$, find an integer k such that $h(X) = g(X)^k$ in $GF(p^n)$. Finding such a k is believed to be very hard in most situations.

3.11.3 LFSR Sequences

We can now explain a phenomenon that is mentioned in Section 5.2 on LFSR sequences.

Suppose that we have a recurrence relation

$$x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \dots + c_{m-1} x_{n+m-1} \pmod{2}.$$

For simplicity, we assume that the associated polynomial

$$P(X) = X^m + c_{m-1}X^{m-1} + c_{m-2}X^{m-2} + \dots + c_0$$

is irreducible mod 2. Then $\mathbf{Z}_2[X] \pmod{P(X)}$ is the field $GF(2^m)$. We regard $GF(2^m)$ as a vector space over \mathbf{Z}_2 with basis $\{1, X, X^2, X^3, \dots, X^{m-1}\}$. Multiplication by X gives a linear transformation of this vector space. Since

$$\begin{aligned} X \cdot 1 &= X, & X \cdot X &= X^2, & X \cdot X^2 &= X^3, & \dots \\ X \cdot X^{m-1} &= X^m \equiv c_0 + c_1 X + \dots + c_{m-1} X^{m-1}, \end{aligned}$$

multiplication by X is represented by the matrix

$$M_X = \begin{pmatrix} 0 & 0 & \cdots & 0 & c_0 \\ 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & c_{m-1} \end{pmatrix}.$$

Suppose we know $(x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m-1})$. We compute

$$\begin{aligned} &(x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m-1}) M_X \\ &= (x_{n+1}, x_{n+2}, x_{n+3}, \dots, c_0 x_n + \dots + c_{m-1} x_{n+m-1}) \\ &\equiv (x_{n+1}, x_{n+2}, x_{n+3}, \dots, x_{n+m}). \end{aligned}$$

Therefore, multiplication by M_X shifts the indices by 1.

It follows easily that multiplication on the right by the

matrix M_X^j sends (x_1, x_2, \dots, x_m) to

$(x_{1+j}, x_{2+j}, \dots, x_{m+j})$. If $M_X^j \equiv I$, the identity matrix, this must be the original vector

(x_1, x_2, \dots, x_m) . Since there are $2^m - 1$ nonzero elements in $GF(2^m)$, it follows from Lagrange's theorem in group theory that $X^{2^m-1} \equiv 1$, which implies that $M_X^{2^m-1} = I$. Therefore, we know that

$$x_1 \equiv x_{2^m}, \quad x_2 \equiv x_{2^m+1}, \dots$$

For any set of initial values (we'll assume that at least one initial value is nonzero), the sequence will repeat after k terms, where k is the smallest positive integer such that $X^k \equiv 1 \pmod{P(X)}$. It can be shown that k divides $2^m - 1$.

In fact, the period of such a sequence is exactly k . This can be proved as follows, using a few results from linear algebra: Let $v = (x_1, \dots, x_m) \neq 0$ be the row vector of initial values. The sequence repeats when $vM_X^j = v$.

This means that the nonzero *row* vector v is in the left null space of the matrix $M_X^j - I$, so

$\det(M_X^j - I) = 0$. But this means that there is a nonzero *column* vector $w = (a_0, \dots, a_{m-1})^T$ in the right null space of $M_X^j - I$. That is, $M_X^j w = w$. Since the matrix M_X^j represents the linear transformation given by multiplication by X^j with respect to the basis $\{1, X, \dots, X^{m-1}\}$, this can be changed back into a relation among polynomials:

$$X^j(a_0 + a_1X + \dots + a_{m-1}X^{m-1}) \equiv a_0 + a_1X + \dots + a_{m-1}X^{m-1} \pmod{P(X)}.$$

But $a_0 + a_1X + \dots + a_{m-1}X^{m-1} \pmod{P(X)}$ is a nonzero element of the field $GF(2^m)$, so we can divide by this element to get $X^j \equiv 1 \pmod{P(X)}$. Since $j = k$ is the first time this happens, the sequence first repeats after k terms, so it has period k .

As mentioned previously, when $2^m - 1$ is prime, all polynomials (except 0 and 1) are generating polynomials for $GF(2^m)$. In particular, X is a generating polynomial and therefore $k = 2^m - 1$ is the period of the recurrence.

3.12 Continued Fractions

There are many situations where we want to approximate a real number by a rational number. For example, we can approximate $\pi = 3.14159265 \dots$ by $314/100 = 157/50$. But $22/7$ is a slightly better approximation, and it is more efficient in the sense that it uses a smaller denominator than $157/50$. The method of continued fractions is a procedure that yields this type of good approximations. In this section, we summarize some basic facts. For proofs and more details, see, for example, [Hardy-Wright], [Niven et al.], [Rosen], and [KraftW].

An easy way to approximate a real number x is to take the largest integer less than or equal to x . This is often denoted by $[x]$. For example, $[\pi] = 3$. If we want to get a better approximation, we need to look at the remaining fractional part. For $\pi = 3.14159 \dots$, this is $.14159 \dots$. This looks close to $1/7 = .142857 \dots$.

One way to express this is to look at

$1/.14159 = 7.06251$. We can approximate this last number by $[7.06251 \dots] = 7$ and therefore conclude that $1/7$ is indeed a good approximation for $.14159$ and that $22/7$ is a good approximation for π . Continuing in this manner yields even better approximations. For example, the next step is to compute

$1/.06251 = 15.9966$ and then take the greatest integer to get 15 (yes, 16 is closer, but the algorithm corrects for this in the next step). We now have

$$\pi \approx 3 + \frac{1}{7 + \frac{1}{15}} = \frac{333}{106}.$$

If we continue one more step, we obtain

$$\pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1}}} = \cfrac{355}{113}.$$

This last approximation is very accurate:

$$\pi = 3.14159265 \dots, \text{ and } 355/113 = 3.14159292 \dots.$$

This procedure works for arbitrary real numbers. Start with a real number x . Let $a_0 = [x]$ and $x_0 = x$. Then (if $x_i \neq a_i$; otherwise, stop) define

$$x_{i+1} = \cfrac{1}{x_i - a_i}, \quad a_{i+1} = [x_{i+1}].$$

We obtain the approximations

$$\cfrac{p_n}{q_n} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\dots + \cfrac{1}{a_n}}}}.$$

We have therefore produced a sequence of rational numbers $p_1/q_1, p_2/q_2, \dots$. It can be shown that each rational number p_k/q_k gives a better approximation to x than any of the preceding rational numbers p_j/q_j with $1 \leq j < k$. Moreover, the following holds.

Theorem

If $|x - (r/s)| < 1/2s^2$ for integers r, s , then $r/s = p_i/q_i$ for some i .

For example, $|\pi - 22/7| \approx .001 < 1/98$ and $22/7 = p_2/q_2$.

Continued fractions yield a convenient way to recognize rational numbers from their decimal expansions. For example, suppose we encounter the decimal 3.764705882 and we suspect that it is the beginning of the decimal expansion of a rational number with small

denominator. The first few terms of the continued fraction are

$$3 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{4 + \cfrac{1}{9803921}}}}.$$

The fact that 9803921 is large indicates that the preceding approximation is quite good, so we calculate

$$3 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{4}}} = \cfrac{64}{17} = 3.7647058823529 \dots,$$

which agrees with all of the terms of the original 3.764605882. Therefore, 64/17 is a likely candidate for the answer. Note that if we had included the 9803921, we would have obtained a fraction that also agrees with the original decimal expansion but has a significantly larger denominator.

Now let's apply the procedure to 12345/11111. We have

$$\cfrac{12345}{11111} = 1 + \cfrac{1}{9 + \cfrac{1}{246 + \cfrac{1}{1 + \cfrac{1}{4}}}}.$$

This yields the numbers

$$1, \quad \cfrac{10}{9}, \quad \cfrac{2461}{2215}, \quad \cfrac{2471}{2224}, \quad \cfrac{12345}{11111}.$$

Note that the numbers 1, 9, 246, 1, 4 are the quotients obtained during the computation of $\gcd(12345, 11111)$ in Subsection 3.1.3 (see Exercise 49).

Calculating the fractions such as

$$\frac{2461}{2215} = 1 + \cfrac{1}{9 + \cfrac{1}{246}}$$

can become tiresome when done in the straightforward way. Fortunately, there is a faster method. Define

$$\begin{aligned} p_{-2} &= 0, & p_{-1} &= 1, & q_{-2} &= 1, & q_{-1} &= 0, \\ p_{n+1} &= n+1 p_n + p_{n-1} \\ q_{n+1} &= a_{n+1} q_n + q_{n-1}. \end{aligned}$$

Then

$$\frac{p_n}{q_n} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots + \cfrac{1}{a_n}}}}.$$

Using these relations, we can compute the partial quotients p_n/q_n from the previous ones, rather than having to start a new computation every time a new a_n is found.

3.13 Exercises

1.
 1. Find integers x and y such that $17x + 101y = 1$.
 2. Find $17^{-1} \pmod{101}$.
2.
 1. Using the identity $x^3 + y^3 = (x + y)(x^2 - xy + y^2)$, factor $2^{333} + 1$ into a product of two integers greater than 1.
 2. Using the congruence $2^2 \equiv 1 \pmod{3}$, deduce that $2^{232} \equiv 1 \pmod{3}$ and show that $2^{333} + 1$ is a multiple of 3.
3.
 1. Solve $7d \equiv 1 \pmod{30}$.
 2. Suppose you write a message as a number $m \pmod{31}$. Encrypt m as $m^7 \pmod{31}$. How would you decrypt?
(Hint: Decryption is done by raising the ciphertext to a power mod 31. Fermat's theorem will be useful.)
4. Solve $5x + 2 \equiv 3x - 7 \pmod{31}$.
5.
 1. Find all solutions of $12x \equiv 28 \pmod{236}$.
 2. Find all solutions of $12x \equiv 30 \pmod{236}$.
6.
 1. Find all solutions of $4x \equiv 20 \pmod{50}$.
 2. Find all solutions of $4x \equiv 21 \pmod{50}$.
7.
 1. Let $n \geq 2$. Show that if n is composite then n has a prime factor $p \leq \sqrt{n}$.
 2. Use the Euclidean algorithm to compute $\gcd(30030, 257)$.
 3. Using the result of parts (a) and (b) and the fact that $30030 = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$, show that 257 is prime.
(Remark: This method of computing one gcd, rather than doing several trial divisions (by 2, 3, 5, ...), is often faster for checking whether small primes divide a number.)
8. Compute $\gcd(12345678987654321, 100)$.

9. 1. Compute $\gcd(4883, 4369)$.
 2. Factor 4883 and 4369 into products of primes.
10. 1. What is $\gcd(111, 11)$? Using the Extended Euclidean algorithm, find $11^{-1} \pmod{111}$.
 2. What is $\gcd(1111, 11)$? Does $11^{-1} \pmod{1111}$ exist?
 3. Find $\gcd(x, 11)$, where x consists of n repeated 1s.
 What can you say about $11^{-1} \pmod{x}$ as a function of n ?
11. 1. Let $F_1 = 1$, $F_2 = 1$, $F_{n+1} = F_n + F_{n-1}$ define the Fibonacci numbers 1, 1, 2, 3, 5, 8, Use the Euclidean algorithm to compute $\gcd(F_n, F_{n-1})$ for all $n \geq 1$.
 2. Find $\gcd(11111111, 11111)$.
 3. Let $a = 111 \cdots 11$ be formed with F_n repeated 1's and let $b = 111 \cdots 11$ be formed with F_{n-1} repeated 1's.
 Find $\gcd(a, b)$. (Hint: Compare your computations in parts (a) and (b).)
12. Let $n \geq 2$. Show that none of the numbers $n! + 2, n! + 3, n! + 4, \dots, n! + n$ are prime.
13. 1. Let p be prime. Suppose a and b are integers such that $ab \equiv 0 \pmod{p}$. Show that either $a \equiv 0$ or $b \equiv 0 \pmod{p}$.
 2. Show that if a, b, n are integers with $n|ab$ and $\gcd(a, n) = 1$, then $n|b$.
14. Let p be prime.
 1. Show that if $x^2 \equiv 0 \pmod{p}$, then $x \equiv 0 \pmod{p}$.
 2. Show that if $k \geq 2$, then $x^2 \equiv 0 \pmod{p^k}$ has solutions with $x \not\equiv 0 \pmod{p^k}$.
15. Let $p \geq 3$ be prime. Show that the only solutions to $x^2 \equiv 1 \pmod{p}$ are $x \equiv \pm 1 \pmod{p}$. (Hint: Apply Exercise 13(a) to $(x+1)(x-1)$.)
16. Find x with $x \equiv 3 \pmod{5}$ and $x \equiv 9 \pmod{11}$.
17. Suppose $x \equiv 2 \pmod{7}$ and $x \equiv 3 \pmod{10}$. What is x congruent to mod 70?
18. Find x with $2x \equiv 1 \pmod{7}$ and $4x \equiv 2 \pmod{9}$. (Hint: Replace $2x \equiv 1 \pmod{7}$ with $x \equiv a \pmod{7}$ for a suitable a ,

and similarly for the second congruence.)

19. A group of people are arranging themselves for a parade. If they line up three to a row, one person is left over. If they line up four to a row, two people are left over, and if they line up five to a row, three people are left over. What is the smallest possible number of people? What is the next smallest number? (Hint: Interpret this problem in terms of the Chinese remainder theorem.)
20. You want to find x such that when you divide x by each of the numbers from 2 to 10, the remainder is 1. The smallest such x is $x = 1$. What is the next smallest x ? (The answer is less than 3000.)
21.
 1. Find all four solutions to $x^2 \equiv 133 \pmod{143}$. (Note that $143 = 11 \cdot 13$.)
 2. Find all solutions to $x^2 \equiv 77 \pmod{143}$. (There are only two solutions in this case. This is because $\gcd(77, 143) \neq 1$.)
22. You need to compute $123456789^{65537} \pmod{581859289607}$. A friend offers to help: 1 cent for each multiplication mod 581859289607 . Your friend is hoping to get more than \$650. Describe how you can have the friend do the computation for less than 25 cents. (Note: $65537 = 2^{16} + 1$ is the most commonly used RSA encryption exponent.)
23. Divide 2^{10203} by 101. What is the remainder?
24. Divide $3^{987654321}$ by 11. What is the remainder?
25. Find the last 2 digits of 123^{562} .
26. Let $p \equiv 3 \pmod{4}$ be prime. Show that $x^2 \equiv -1 \pmod{p}$ has no solutions. (Hint: Suppose x exists. Raise both sides to the power $(p-1)/2$ and use Fermat's theorem. Also, $(-1)^{(p-1)/2} = -1$ because $(p-1)/2$ is odd.)
27. Let p be prime. Show that $a^p \equiv a \pmod{p}$ for all a .
28. Let p be prime and let a and b be integers. Show that $(a+b)^p \equiv a^p + b^p \pmod{p}$.
29.
 1. Evaluate $7^7 \pmod{4}$.
 2. Use part (a) to find the last digit of 7^{7^7} . (Note: a^{bc} means $a^{(bc)}$ since the other possible interpretation would be $(a^b)^c = a^{bc}$, which is written more easily without a second exponentiation.) (Hint: Use part (a) and the Basic Principle that follows Euler's Theorem.)
30. You are told that exactly one of the numbers

$$2^{1000} + 277, \quad 2^{1000} + 291, \quad 2^{1000} + 297$$

is prime and you have one minute to figure out which one. Describe calculations you could do (with software such as MATLAB or Mathematica) that would give you a very good chance of figuring out which number is prime? Do not do the calculations. Do not try to factor the numbers. They do not have any prime factors less than 10^9 . You may use modular exponentiation, but you may not use commands of the form “IsPrime[n]” or “NextPrime[n].” (See Computer Problem 3 below.)

31. 1. Let $p = 7, 13$, or 19 . Show that $a^{1728} \equiv 1 \pmod{p}$ for all a with $p \nmid a$.
2. Let $p = 7, 13$, or 19 . Show that $a^{1729} \equiv a \pmod{p}$ for all a . (Hint: Consider the case $p|a$ separately.)
3. Show that $a^{1729} \equiv a \pmod{1729}$ for all a . Composite numbers n such that $a^n \equiv a \pmod{n}$ for all a are called Carmichael numbers. They are rare (561 is another example), but there are infinitely many of them [Alford et al. 2].
32. 1. Show that $2^{10} \equiv 1 \pmod{11}$ and $2^5 \equiv 1 \pmod{31}$.
2. Show that $2^{340} \equiv 1 \pmod{341}$.
3. Is 341 prime?
33. 1. Let p be prime and let $a \not\equiv 0 \pmod{p}$. Let $b \equiv a^{p-2} \pmod{p}$. Show that $ab \equiv 1 \pmod{p}$.
2. Use the method of part (a) to solve $2x \equiv 1 \pmod{7}$.
34. You are appearing on the Math Superstars Show and, for the final question, you are given a 500-digit number n and are asked to guess whether or not it is prime. You are told that n is either prime or the product of a 200-digit prime and a 300-digit prime. You have one minute, and fortunately you have a computer. How would you make a guess that's very probably correct? Name any theorems that you are using.
35. 1. Compute $\phi(d)$ for all of the divisors of 10 (namely, $1, 2, 5, 10$), and find the sum of these $\phi(d)$.
2. Repeat part (a) for all of the divisors of 12 .
3. Let $n \geq 1$. Conjecture the value of $\sum \phi(d)$, where the sum is over the divisors of n . (This result is proved in many elementary number theory texts.)

36. Find a number $\alpha \pmod{7}$ that is a primitive root mod 7 and find a number $\gamma \not\equiv 0 \pmod{7}$ that is not a primitive root mod 7.
Show that α and γ have the desired properties.

- 37.
1. Show that every nonzero congruence class mod 11 is a power of 2, and therefore 2 is a primitive root mod 11.
 2. Note that $2^3 \equiv 8 \pmod{11}$. Find x such that $8^x \equiv 2 \pmod{11}$. (Hint: What is the inverse of 3 (mod 10)?)
 3. Show that every nonzero congruence class mod 11 is a power of 8, and therefore 8 is a primitive root mod 11.
 4. Let p be prime and let α be a primitive root mod p . Let $h \equiv \alpha^y \pmod{p}$ with $\gcd(y, p-1) = 1$. Let $xy \equiv 1 \pmod{p-1}$. Show that $h^x \equiv \alpha \pmod{p}$.
 5. Let p and h be as in part (d). Show that h is a primitive root mod p . (Remark: Since there are $\phi(p-1)$ possibilities for the exponent x in part (d), this yields all of the $\phi(p-1)$ primitive roots mod p .)
 6. Use the method of part (e) to find all primitive roots for $p = 13$, given that 2 is a primitive root.

38. It is known that 14 is a primitive root for the prime $p = 30000001$. Let $b \equiv 14^{9000000} \pmod{p}$. (The exponent is $3(p-1)/10$.)

1. Explain why $b^{10} \equiv 1 \pmod{p}$.
2. Explain why $b \not\equiv 1 \pmod{p}$.

39.

1. Find the inverse of $\begin{pmatrix} 1 & 1 \\ 6 & 1 \end{pmatrix} \pmod{26}$.

2. Find all values of $b \pmod{26}$ such that $\begin{pmatrix} 1 & 1 \\ b & 1 \end{pmatrix} \pmod{26}$ is invertible.

40. Find the inverse of $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \pmod{2}$.

41. Find all primes p for which $\begin{pmatrix} 3 & 5 \\ 7 & 3 \end{pmatrix} \pmod{p}$ is not invertible.

- 42.
1. Use the Legendre symbol to show that $x^2 \equiv 5 \pmod{19}$ has a solution.
 2. Use the method of Section 3.9 to find a solution to $x^2 \equiv 5 \pmod{19}$.

43. Use the Legendre symbol to determine which of the following congruences have solutions (each modulus is prime):

1. $X^2 \equiv 123 \pmod{401}$
2. $X^2 \equiv 43 \pmod{179}$
3. $X^2 \equiv 1093 \pmod{65537}$

44. 1. Let n be odd and assume $\gcd(a, n) = 1$. Show that if $\left(\frac{a}{n}\right) = -1$, then a is not a square mod n .

2. Show that $\left(\frac{3}{35}\right) = +1$.

3. Show that 3 is not a square mod 35.

45. Let $n = 15$. Show that $\left(\frac{2}{n}\right) \not\equiv 2^{(n-1)/2} \pmod{n}$.

46. 1. Show that $\left(\frac{3}{65537}\right) = -1$.

2. Show that $3^{(65537-1)/2} \not\equiv 1 \pmod{65537}$.

3. Use the procedure of [Exercise 54](#) to show that 3 is a primitive root mod 65537. (Remark: The same proof shows that 3 is a primitive root for any prime $p \geq 5$ such that $p - 1$ is a power of 2. However, there are only six known primes with $p - 1$ a power of 2; namely, 2, 3, 5, 17, 257, 65537. They are called *Fermat primes*.)

47. 1. Show that the only irreducible polynomials in $\mathbf{Z}_2[X]$ of degree at most 2 are X , $X + 1$, and $X^2 + X + 1$.

2. Show that $X^4 + X + 1$ is irreducible in $\mathbf{Z}_2[X]$. (Hint: If it factors, it must have at least one factor of degree at most 2.)

3. Show that $X^4 \equiv X + 1$, $X^8 \equiv X^2 + 1$, and $X^{16} \equiv X \pmod{X^4 + X + 1}$.

4. Show that $X^{15} \equiv 1 \pmod{X^4 + X + 1}$.

48. 1. Show that $X^2 + 1$ is irreducible in $\mathbf{Z}_3[X]$.

2. Find the multiplicative inverse of $1 + 2X$ in $\mathbf{Z}_3[X] \pmod{X^2 + 1}$.

49. Show that the quotients in the Euclidean algorithm for $\gcd(a, b)$ are exactly the numbers a_0, a_1, \dots that appear in the continued

fraction of a/b .

50. 1. Compute several steps of the continued fractions of $\sqrt{3}$ and $\sqrt{7}$. Do you notice any patterns? (It can be shown that the a_i 's in the continued fraction of every irrational number of the form $a + b\sqrt{d}$ with a, b, d rational and $d > 0$ eventually become periodic.)
2. For each of $d = 3, 7$, let n be such that $a_{n+1} = 2a_0$ in the continued fraction of \sqrt{d} . Compute p_n and q_n and show that $x = p_n$ and $y = q_n$ give a solution of what is known as Pell's equation: $x^2 - dy^2 = 1$.
3. Use the method of part (b) to solve $x^2 - 19y^2 = 1$.
51. Compute several steps of the continued fraction expansion of e . Do you notice any patterns? (On the other hand, the continued fraction expansion of π seems to be fairly random.)
52. Compute several steps of the continued fraction expansion of $(1 + \sqrt{5})/2$ and compute the corresponding numbers p_n and q_n (defined in [Section 3.12](#)). The sequences p_0, p_1, p_2, \dots and q_1, q_2, \dots are what famous sequence of numbers?
53. Let a and $n > 1$ be integers with $\gcd(a, n) = 1$. The **order** of $a \bmod n$ is the smallest positive integer r such that $a^r \equiv 1 \pmod{n}$. We denote $r = \text{ord}_n(a)$.
1. Show that $r \leq \phi(n)$.
 2. Show that if $m = rk$ is a multiple of r , then $a^m \equiv 1 \pmod{n}$.
 3. Suppose $a^t \equiv 1 \pmod{n}$. Write $t = qr + s$ with $0 \leq s < r$ (this is just division with remainder). Show that $a^s \equiv 1 \pmod{n}$.
 4. Using the definition of r and the fact that $0 \leq s < r$, show that $s = 0$ and therefore $r|t$. This, combined with part (b), yields the result that $a^t \equiv 1 \pmod{n}$ if and only if $\text{ord}_n(a)|t$.
 5. Show that $\text{ord}_n(a)|\phi(n)$.
54. This exercise will show by example how to use the results of [Exercise 53](#) to prove a number is a primitive root mod a prime p , once we know the factorization of $p - 1$. In particular, we'll show that 7 is a primitive root mod 601. Note that $600 = 2^3 \cdot 3 \cdot 5^2$.
1. Show that if an integer $r < 600$ divides 600, then it divides at least one of 300, 200, 120 (these numbers are $600/2$, $600/3$, and $600/5$).

2. Show that if $\text{ord}_{601}(7) < 600$, then it divides one of the numbers 300, 200, 120.

3. A calculation shows that

$$7^{300} \equiv 600, \quad 7^{200} \equiv 576, \quad 7^{120} \equiv 423 \pmod{601}.$$

Why can we conclude that $\text{ord}_{601}(7)$ does not divide 300, 200, or 120?

4. Show that 7 is a primitive root mod 601.

5. In general, suppose p is a prime and $p - 1 = q_1^{a_1} \cdots q_s^{a_s}$ is the factorization of $p - 1$ into primes. Describe a procedure to check whether a number α is a primitive root mod p . (Therefore, if we need to find a primitive root mod p , we can simply use this procedure to test the numbers $\alpha = 2, 3, 5, 6, \dots$ in succession until we find one that is a primitive root.)

55. We want to find an exponent k such that $3^k \equiv 2 \pmod{65537}$.

1. Observe that $2^{32} \equiv 1 \pmod{65537}$, but $2^{16} \not\equiv 1 \pmod{65537}$. It can be shown ([Exercise 46](#)) that 3 is a primitive root mod 65537, which implies that $3^n \equiv 1 \pmod{65537}$ if and only if $65536|n$. Use this to show that $2048|k$ but 4096 does not divide k . (Hint: Raise both sides of $3^k \equiv 2$ to the 16th and to the 32nd powers.)

2. Use the result of part (a) to conclude that there are only 16 possible choices for k that need to be considered. Use this information to determine k . This problem shows that if $p - 1$ has a special structure, for example, a power of 2, then this can be used to avoid exhaustive searches. Therefore, such primes are cryptographically weak. See [Exercise 12 in Chapter 10](#) for a reinterpretation of the present problem.

56. 1. Let $x = b_1 b_2 \dots b_w$ be an integer written in binary (for example, when $x = 1011$, we have $b_1 = 1, b_2 = 0, b_3 = 1, b_4 = 1$). Let y and n be integers. Perform the following procedure:

1. Start with $k = 1$ and $s_1 = 1$.

2. If $b_k = 1$, let $r_k \equiv s_k y \pmod{n}$. If $b_k = 0$, let $r_k = s_k$.

3. Let $s_{k+1} \equiv r_k^2 \pmod{n}$.

4. If $k = w$, stop. If $k < w$, add 1 to k and go to (2).

Show that $r_w \equiv y^x \pmod{n}$.

2. Let x , y , and n be positive integers. Show that the following procedure computes $y^x \pmod{n}$.

1. Start with $a = x$, $b = 1$, $c = y$.
2. If a is even, let $a = a/2$, and let $b = b$, $c \equiv c^2 \pmod{n}$.
3. If a is odd, let $a = a - 1$, and let $b \equiv bc \pmod{n}$, $c = c$.
4. If $a \neq 0$, go to step 2.
5. Output b .

(Remark: This algorithm is similar to the one in part (a), but it uses the binary bits of x in reverse order.)

57. Here is how to construct the x guaranteed by the general form of the Chinese remainder theorem. Suppose m_1, \dots, m_k are integers with $\gcd(m_i, m_j) = 1$ whenever $i \neq j$. Let a_1, \dots, a_k be integers. Perform the following procedure:

1. For $i = 1, \dots, k$, let $z_i = m_1 \cdots m_{i-1} m_{i+1} \cdots m_k$.
2. For $i = 1, \dots, k$, let $y_i \equiv z_i^{-1} \pmod{m_i}$.
3. Let $x = a_1 y_1 z_1 + \cdots + a_k y_k z_k$.

Show $x \equiv a_i \pmod{m_i}$ for all i .

58. Alice designs a cryptosystem as follows (this system is due to Rabin). She chooses two distinct primes p and q (preferably, both p and q are congruent to 3 mod 4) and keeps them secret. She makes $n = pq$ public. When Bob wants to send Alice a message m , he computes $x \equiv m^2 \pmod{n}$ and sends x to Alice. She makes a decryption machine that does the following: When the machine is given a number x , it computes the square roots of x mod n since it knows p and q . There is usually more than one square root. It chooses one at random, and gives it to Alice. When Alice receives x from Bob, she puts it into her machine. If the output from the machine is a meaningful message, she assumes it is the correct message. If it is not meaningful, she puts x into the machine again. She continues until she gets a meaningful message.

1. Why should Alice expect to get a meaningful message fairly soon?

2. If Oscar intercepts x (he already knows n), why should it be hard for him to determine the message m ?
3. If Eve breaks into Alice's office and thereby is able to try a few chosen-ciphertext attacks on Alice's decryption machine, how can she determine the factorization of n ?
59. This exercise shows that the Euclidean algorithm computes the gcd. Let a, b, q_i, r_i be as in Subsection 3.1.3.
1. Let d be a common divisor of a, b . Show that $d|r_1$, and use this to show that $d|r_2$.
 2. Let d be as in (a). Use induction to show that $d|r_i$ for all i . In particular, $d|r_k$, the last nonzero remainder.
 3. Use induction to show that $r_k|r_i$ for $1 \leq i \leq k$.
 4. Using the facts that $r_k|r_1$ and $r_k|r_2$, show that $r_k|b$ and then $r_k|a$. Therefore, r_k is a common divisor of a, b .
 5. Use (b) to show that $r_k \geq d$ for all common divisors d , and therefore r_k is the greatest common divisor.
60. Let p and q be distinct primes.
1. Show that among the integers m satisfying $1 \leq m < pq$, there are $q - 1$ multiples of p , and there are $p - 1$ multiples of q .
 2. Suppose $\gcd(m, pq) > 1$. Show that m is a multiple of p or a multiple of q .
 3. Show that if $1 \leq m < pq$, then m cannot be a multiple of both p and q .
 4. Show that the number of integers m with $1 \leq m < q$ such that $\gcd(m, pq) = 1$ is $pq - 1 - (p - 1) - (q - 1) = (p - 1)(q - 1)$.
(Remark: This proves the formula that $\phi(pq) = (p - 1)(q - 1)$.)
- 61.
1. Give an example of integers $m \neq n$ with $\gcd(m, n) > 1$ and integers a, b such that the simultaneous congruences
- $$x \equiv a \pmod{m}, \quad x \equiv b \pmod{n}$$
- have no solution.
2. Give an example of integers $m \neq n$ with $\gcd(m, n) > 1$ and integers $a \neq b$ such that the simultaneous congruences

$$x \equiv a \pmod{m}, \quad x \equiv b \pmod{n}$$

have a solution.

3.14 Computer Problems

1. Evaluate $\gcd(8765, 23485)$.
2.
 1. Find integers x and y with $65537x + 3511y = 1$.
 2. Find integers x and y with $65537x + 3511y = 17$.

3. You are told that exactly one of the numbers

$$2^{1000} + 277, \quad 2^{1000} + 291, \quad 2^{1000} + 297$$

is prime and you have one minute to figure out which one. They do not have any prime factors less than 10^9 . You may use modular exponentiation, but you may not use commands of the form “`IsPrime[n]`” or “`NextPrime[n]`.” (This makes explicit [Exercise 30](#) above.)

4. Find the last five digits of $3^{1234567}$. (Note: Don’t ask the computer to print $3^{1234567}$. It is too large!)
5. Look at the decimal expansion of
 $e = 2.71828182845904523 \dots$. Find the consecutive digits 71, the consecutive digits 271, and the consecutive digits 4523 form primes. Find the first set of five consecutive digits that form a prime (04523 does not count as a five-digit number).
6. Solve $314x \equiv 271 \pmod{11111}$.
7. Find all solutions to $216x \equiv 66 \pmod{606}$.
8. Find an integer such that when it is divided by 101 the remainder is 17, when it is divided by 201 the remainder is 18, and when it is divided by 301 the remainder is 19.
9. Let $n = 391 = 17 \cdot 23$. Show that $2^{n-1} \not\equiv 1 \pmod{n}$. Find an exponent $j > 0$ such that $2^j \equiv 1 \pmod{n}$.
10. Let $n = 84047 \cdot 65497$. Find x and y with $x^2 \equiv y^2 \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$.

11. Let $M = \begin{matrix} 1 & 2 & 4 \\ 1 & 5 & 25 \\ 1 & 14 & 196 \end{matrix}$.

1. Find the inverse of $M \pmod{101}$.
2. For which primes p does M not have an inverse mod p ?

12. Find the square roots of 26055 mod the prime 34807 .
13. Find all square roots of 1522756 mod 2325781 .
14. Try to find a square root of 48382 mod the prime 83987 , using the method of [Section 3.9](#). Square your answer to see if it is correct.
What number did you find the square root of?

Chapter 4 The One-Time Pad

The one-time pad, which is an unbreakable cryptosystem, was described by Frank Miller in 1882 as a means of encrypting telegrams. It was rediscovered by Gilbert Vernam and Joseph Mauborgne around 1918. In terms of security, it is the best possible system, but implementation makes it unsuitable for most applications.

In this chapter, we introduce the one-time pad and show why a given key should not be used more than once. We then introduce the important concepts of perfect secrecy and ciphertext indistinguishability, topics that have become prominent in cryptography in recent years.

4.1 Binary Numbers and ASCII

In many situations involving computers, it is more natural to represent data as strings of 0s and 1s, rather than as letters and numbers.

Numbers can be converted to binary (or base 2), if desired, which we'll quickly review. Our standard way of writing numbers is in base 10. For example, 123 means $1 \times 10^2 + 2 \times 10^1 + 3$. Binary uses 2 in place of 10 and needs only the digits 0 and 1. For example, 110101 in binary represents $2^5 + 2^4 + 2^2 + 1$ (which equals 53 in base 10).

Each 0 or 1 is called a **bit**. A representation that takes eight bits is called an eight-bit number, or a **byte**. The largest number that 8 bits can represent is 255, and the largest number that 16 bits can represent is 65535.

Often, we want to deal with more than just numbers. In this case, words, symbols, letters, and numbers are given binary representations. There are many possible ways of doing this. One of the standard ways is called ASCII, which stands for American Standard Code for Information Interchange. Each character is represented using seven bits, allowing for 128 possible characters and symbols to be represented. Eight-bit blocks are common for computers to use, and for this reason, each character is often represented using eight bits. The eighth bit can be used for checking parity to see if an error occurred in transmission, or is often used to extend the list of characters to include symbols such as ü and è .

Table 4.1 gives the ASCII equivalents for some standard symbols.

Table 4.1 ASCII Equivalents of Selected Symbols

symbol	!	"	#	\$	%	&	,
decimal	33	34	35	36	37	38	39
binary	0100001	0100010	0100011	0100100	0100101	0100110	0100111
()	*	+	,	-	.	/
40	41	42	43	44	45	46	47
0101000	0101001	0101010	0101011	0101100	0101101	0101110	0101111
0	1	2	3	4	5	6	7
48	49	50	51	52	53	54	55
0110000	0110001	0110010	0110011	0110100	0110101	0110110	0110111
8	9	:	;	<	=	>	?
56	57	58	59	60	61	62	63
0111000	0111001	0111010	0111011	0111100	0111101	0111110	0111111
@	A	B	C	D	E	F	G
64	65	66	67	68	69	70	71
1000000	1000001	1000010	1000011	1000100	1000101	1000110	1000111

Table 4.1 Full Alternative Text

4.2 One-Time Pads

Start by representing the message as a sequence of 0s and 1s. This can be accomplished by writing all numbers in binary, for example, or by using ASCII, as discussed in the previous section. But the message could also be a digitalized video or audio signal.

The key is a random sequence of 0s and 1s of the same length as the message. Once a key is used, it is discarded and never used again. The encryption consists of adding the key to the message mod 2, bit by bit. This process is often called **exclusive or**, and is denoted by *XOR* or \oplus . In other words, we use the rules $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 1 = 0$. For example, if the message is 00101001 and the key is 10101100, we obtain the ciphertext as follows:

(plaintext)	00101001
(key)	\oplus 10101100
(ciphertext)	10000101

Decryption uses the same key. Simply add the key onto the ciphertext: $10000101 \oplus 10101100 = 00101001$.

A variation is to leave the plaintext as a sequence of letters. The key is then a random sequence of shifts, each one between 0 and 25. Decryption uses the same key, but subtracts instead of adding the shifts.

This encryption method is completely unbreakable for a ciphertext-only attack. For example, suppose the ciphertext is *FIOWPSLQNTISJQL*. The plaintext could be *we will win the war* or it could be *the duck wants out*. Each one is possible, along with all other messages of the same length. Therefore the ciphertext gives no information about the plaintext (except for its length).

This will be made more precise in [Section 4.4](#) and when we discuss Shannon's theory of entropy in [Chapter 20](#).

If we have a piece of the plaintext, we can find the corresponding piece of the key, but it will tell us nothing about the remainder of the key. In most cases a chosen plaintext or chosen ciphertext attack is not possible. But such an attack would only reveal the part of the key used during the attack, which would not be useful unless this part of the key were to be reused.

How do we implement this system, and where can it be used? The key can be generated in advance. Of course, there is the problem of generating a truly random sequence of 0s and 1s. One way would be to have some people sitting in a room flipping coins, but this would be too slow for most purposes. It is often suggested that we could take a Geiger counter and count how many clicks it makes in a small time period, recording a 0 if this number is even and 1 if it is odd, but care must be taken to avoid biases (see [Exercise 12 in Chapter 5](#)). There are other ways that are faster but not quite as random that can be used in practice (see [Chapter 5](#)); but it is easy to see that quickly generating a good key is difficult. Once the key is generated, it can be sent by a trusted courier to the recipient. The message can then be sent when needed. It is reported that the “hot line” between Washington, D.C., and Moscow used one-time pads for secure communications between the leaders of the United States and the U.S.S.R. during the Cold War.

A disadvantage of the one-time pad is that it requires a very long key, which is expensive to produce and expensive to transmit. Once the key is used up, it is dangerous to reuse it for a second message; any knowledge of the first message gives knowledge of the second, for example. Therefore, in most situations, various methods are used in which a small input can generate a reasonably random sequence of 0s and 1s,

hence an “approximation” to a one-time pad. The amount of information carried by the courier is then several orders of magnitude smaller than the messages that will be sent. Two such methods, which are fast but not highly secure, are described in [Chapter 5](#).

A variation of the one-time pad has been developed by Maurer, Rabin, Ding, and others. Suppose it is possible to have a satellite produce and broadcast several random sequences of bits at a rate fast enough that no computer can store more than a very small fraction of the outputs. Alice wants to send a message to Bob. They use a public key method such as RSA (see [Chapter 9](#)) to agree on a method of sampling bits from the random bit streams. Alice and Bob then use these bits to generate a key for a one-time pad. By the time Eve has decrypted the public key transmission, the random bits collected by Alice and Bob have disappeared, so Eve cannot decrypt the message. In fact, since the encryption used a one-time pad, she can never decrypt it, so Alice and Bob have achieved everlasting security for their message. Note that bounded storage is an integral assumption for this procedure. The production and the accurate sampling of the bit streams are also important implementation issues.

4.3 Multiple Use of a One-Time Pad

Alice sends messages to Bob, Carla, and Dante. She encrypts each message with a one-time pad, but she's lazy and uses the same key for each message. In this section, we'll show how Eve can decrypt all three messages.

Suppose the messages are M_1, M_2, M_3 and the key is K . The ciphertexts C_1, C_2, C_3 are computed as $M_i \oplus K = C_i$. Eve computes

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2.$$

Similarly, she obtains $M_1 \oplus M_3 = C_1 \oplus C_3$ and $M_2 \oplus M_3 = C_2 \oplus C_3$. The key K has disappeared, and Eve's task is to deduce M_1, M_2, M_3 from knowledge of $M_1 \oplus M_2, M_1 \oplus M_3, M_2 \oplus M_3$. The following example shows some basic ideas of the method.

Let's assume for simplicity that the messages are written in capital letters with spaces but with no other punctuation. The letters are converted to ASCII using

$$A = 1000001, B = 1000010, \dots, Z = 1011010, \text{ space} = 0100000$$

(the letters A to Z are the numbers 65 through 90 written in binary, and *space* is 32 in binary; see Table 4.1).

The XORs of the messages are

$M_1 \oplus M_2 =$
0000000 0001100 1100100 0001101 1100001 0011110 1111001 1100001
1100100 0011100 0001001 0001111 0010011 0010111 0000111

$M_1 \oplus M_3 =$
0000001 1100101 1100001 0000101 1100001 0011011 1100101 0010011
1100101 1110011 0010001 0001100 0010110 0000100 0001101

$M_2 \oplus M_3 =$
0000001 1101001 0000101 0001000 0000000 0000101 0011100 1110010
0000001 1101111 0011000 0000011 0000101 0010011 0001010.

Note that the first block of $M_1 \oplus M_2$ is 0000000. This means that the first letter of M_1 is the same as the first letter of M_2 . But the observation that makes the biggest difference for us is that “space” has a 0 as its leading bit, while all the letters have 1 as the leading bit. Therefore, if the leading bit of an XOR block is 1, then it arises from a letter XORed with “space.”

For example, the third block of $M_1 \oplus M_2$ is 1100100 and the third block of $M_1 \oplus M_3$ is 1100001. These can happen only from “space \oplus 1000100” and “space \oplus 1000001”, respectively. It follows easily that M_1 has “space” as its third entry, M_2 has 1000100 = H as its third entry, and M_3 has A as its third entry. Similarly, we obtain the 2nd, 7th, 8th, and 9th entries of each message.

The 5th entries cause a little more trouble: $M_1 \oplus M_2$ and $M_1 \oplus M_3$ tell us that there is “space” XORed with A , and $M_2 \oplus M_3$ tells us that the 5th entries of M_2 and M_3 are equal, but we could have “space $A A$ ” or “ A space space.” We need more information from surrounding letters to determine which it is.

To proceed, we use the fact that some letters are more common than others, and therefore certain XORs are more likely to arise from these than from others. For example, the block 0010001 is more likely to arise from $E \oplus T = 1000101 \oplus 1010100$ than from $K \oplus Z = 1001011 \oplus 1011010$. The most common

letters in English are

$E, T, A, O, I, N, S, H, R, D, L$. Make a table of the XORs of each pair of these letters. If an XOR block occurs in this table, we guess that it comes from one of the pairs yielding this block. For example, 0001001 arises from $A \oplus H$ and from $E \oplus L$, so we guess that the 11th block of $M_1 \oplus M_2$ comes from one of these two combinations. This might not be a correct guess, but we expect such guesses to be right often enough to yield a lot of information.

Rather than produce the whole table, we give the blocks that we need and the combinations of these frequent letters that produce them:

0000001 : $D \oplus E, H \oplus I, R \oplus S$
0000011 : $L \oplus O$
0000100 : $A \oplus E, H \oplus L$
0000101 : $A \oplus D, I \oplus L$
0001000 : $A \oplus I, D \oplus L$
0001001 : $A \oplus H, E \oplus L$
0001100 : $D \oplus H, E \oplus I$
0001111 : $A \oplus N$
0010001 : $E \oplus T$
0010011 : $A \oplus R$
0010110 : $D \oplus R, E \oplus S$
0010111 : $D \oplus S, E \oplus R$
0011000 : $L \oplus T$
0011011 : $H \oplus S, I \oplus R, O \oplus T$
0011110 : $L \oplus R$

Let's look at the next-to-last block of each XOR. In $M_1 \oplus M_2$, we have 0010111, which could come from $D \oplus S$ or $E \oplus R$. In $M_1 \oplus M_3$, we have 0000100, which could come from $A \oplus E$ or $H \oplus L$. In $M_2 \oplus M_3$, we have 0010011, which could come from $A \oplus R$. The only combination consistent with these guesses is E, R, A for the next-to-last letters of M_1, M_2, M_3 , respectively. This type of reasoning, combined with the information obtained from the occurrence of spaces, yields the following progress (- indicates a space, * indicates a letter to be determined):

$$\begin{aligned}M_1 &= *E-**R-A-SE*RE* \\M_2 &= *ID**LY-DOL*AR* \\M_3 &= *-A**IERE-T*DA*\end{aligned}$$

The first letter of M_3 is a one-letter word. The XOR block 0000001 gives us the possibilities D, E, H, I, R, S , so we guess M_3 starts with I . The XORs tell us that M_1 and M_2 start with H .

Now let's look at the 12th letters. The block 0001111 for $M_1 \oplus M_2$ suggests $A \oplus N$. The block for $M_1 \oplus M_3$ suggests $D \oplus H$ and $E \oplus I$. The block for $M_2 \oplus M_3$ suggests $L \oplus O$. These do not have a common solution, so we know that one of the less common letters occurs in at least one of the messages. But M_2 ends with $DOL*AR*$, and a good guess is that this should be $DOLLARS$. If so, then the XOR information tells us that M_1 ends in SECRET, and M_3 ends in TODAY, both of which are words. So this looks like a good guess. Our progress so far is the following:

$$\begin{aligned}M_1 &= HE-**R-A-SECRET \\M_2 &= HID**LY-DOLLARS \\M_3 &= I-A**IERE-TODAY\end{aligned}$$

It is time to revisit the 5th letters. We already know that they are “space $A A$ ” or “ A space space.” The first possibility requires M_1 to have two consecutive one-letter words, which seems unlikely. The second possibility means that M_3 starts with the two words I-A*, so we can guess this is I AM. The XOR information tells us that M_1 and M_2 have H and E , respectively. We now have all the letters:

$$\begin{aligned}M_1 &= HE-HAR-A-SECRET \\M_2 &= HIDE-LY-DOLLARS \\M_3 &= I-AM-IERE-TODAY\end{aligned}$$

Something seems to be wrong in the 6th column. These letters were deduced from the assumption that they all were common letters, and this must have been wrong. But at this point, we can make educated guesses. When we change I to H in M_3 , and make the corresponding

changes in M_1 and M_2 required by the XORs, we get the final answer:

$M_1 = \text{HE-HAS-A-SECRET}$
 $M_2 = \text{HIDE-MY-DOLLARS}$
 $M_3 = \text{I-AM-HERE-TODAY}.$

These techniques can also be used when there are only two messages, but progress is usually slower. More possible combinations must be tried, and more false deductions need to be corrected during the decryption. The process can be automated. See [Dawson-Nielsen].

The use of spaces and the restriction to capital letters made the decryption in the example easier. However, even if spaces are removed and more symbols are used, decryption is usually possible, though much more tedious.

During World War II, problems with production and distribution of one-time pads forced Russian embassies to reuse some keys. This was discovered, and part of the Venona Project by the U.S. Army's Signal Intelligence Service (the predecessor to NSA) was dedicated to deciphering these messages. Information obtained this way revealed several examples of Russian espionage, for example, in the Manhattan Project's development of atomic bombs, in the State Department, and in the White House.

4.4 Perfect Secrecy of the One-Time Pad

Everyone knows that the one-time pad provides perfect secrecy. But what does this mean? In this section, we make this concept precise. Also, we know that it is very difficult in practice to produce a truly random key for a one-time pad. In the next section, we show quantitatively how biases in producing the key affect the security of the encryption.

In [Section 20.4](#), we repeat some of the arguments of the present section and phrase them in terms of entropy and information theory.

The topics of this section and the next are part of the subject known as **Provable Security**. Rather than relying on intuition that a cryptosystem is secure, the goal is to isolate exactly what fundamental problems are the basis for its security. The result of the next section shows that the security of a one-time pad is based on the quality of the random number generator. In [Section 10.5](#), we will show that the security of the ElGamal public key cryptosystem reduces to the difficulty of the Computational Diffie-Hellman problem, one of the fundamental problems related to discrete logarithms. In [Section 12.3](#), we will use the Random Oracle Model to relate the security of a simple cryptosystem to the noninvertibility of a one-way function. Since these fundamental problems have been well studied, it is easier to gauge the security levels of the cryptosystems.

First, we need to define **conditional probability**. Let's consider an example. We know that if it rains Saturday, then there is a reasonable chance that it will rain on Sunday. To make this more precise, we want to compute

the probability that it rains on Sunday, given that it rains on Saturday. So we restrict our attention to only those situations where it rains on Saturday and count how often this happens over several years. Then we count how often it rains on both Saturday and Sunday. The ratio gives an estimate of the desired probability. If we call A the event that it rains on Saturday and B the event that it rains on Sunday, then the **conditional probability of B given A** is

$$P(B | A) = \frac{P(A \cap B)}{P(A)},$$

(4.1)

where $P(A)$ denotes the probability of the event A . This formula can be used to define the conditional probability of one event given another for any two events A and B that have probabilities (we implicitly assume throughout this discussion that any probability that occurs in a denominator is nonzero).

Events A and B are **independent** if

$P(A \cap B) = P(A) P(B)$. For example, if Alice flips a fair coin, let A be the event that the coin ends up Heads. If Bob rolls a fair six-sided die, let B be the event that he rolls a 3. Then $P(A) = 1/2$ and $P(B) = 1/6$. Since all 12 combinations of $\{\text{Head}, \text{Tail}\}$ and $\{1, 2, 3, 4, 5, 6\}$ are equally likely, $P(A \cap B) = 1/12$, which equals $P(A) P(B)$. Therefore, A and B are independent.

If A and B are independent, then

$$P(B | A) = \frac{P(A \cap B)}{P(A)} = \frac{P(A) P(B)}{P(A)} = P(B),$$

which means that knowing that A happens does not change the probability that B happens. By reversing the steps in the above equation, we see that

$$A \text{ and } B \text{ are independent} \iff P(B | A) = P(B).$$

An example of events that are not independent is the original example, where A is the event that it rains on Saturday and B is the event that it rains on Sunday, since $P(B | A) > P(B)$. (Unfortunately, a widely used high school algebra text published around 2005 gave exactly one example of independent events: A and B .)

How does this relate to cryptography? In a cryptosystem, there is a set of possible keys. Let's say we have N keys. If we have a perfect random number generator to choose the keys, then the probability that the key is k is $P(K = k) = 1/N$. In this case we say that the key is chosen uniformly randomly. In any case, we assume that each key has a certain probability of being chosen. We also have various possible plaintexts m and each one has a certain probability $P(M = m)$. These probably do not all have the same probability. For example, the message *attack at noon* is usually more probable than *two plus two equals seven*. Finally, each possible ciphertext c has a probability $P(C = c)$.

We say that a cryptosystem has **perfect secrecy** if

$$P(M = m | C = c) = P(M = m)$$

for all possible plaintexts m and all possible ciphertexts c . In other words, knowledge of the ciphertext never changes the probability that a given plaintext occurs. This means that eavesdropping gives no advantage to Eve if she wants to guess the message.

We can now formalize what we claimed about the one-time pad.

Theorem

If the key is chosen uniformly randomly, then the one-time pad has perfect secrecy.

Proof. We need to show that

$$P(M = m \mid C = c) = P(M = m) \text{ for each pair } m, c.$$

Let's say that there are N keys, each of which has probability $1/N$. We start by showing that each possible ciphertext c also has probability $1/N$. Start with any plaintext m . If c is the ciphertext, then the key is $k = m \oplus c$. Therefore, the probability that c is the ciphertext is the probability that $m \oplus c$ is the key, namely $1/N$, since all keys have this probability. Therefore, we have proved that

$$P(C = c \mid M = m) = 1/N$$

for each c and m .

We now combine the contributions from the various possibilities for m . Note that if we sum $P(M = m)$ over all possible m , then

$$\sum_m P(M = m) = 1$$

(4.2)

since this is the probability of the plaintext existing.

Similarly, the event $C = c$ can be split into the disjoint sets $(C = c \cap M = m)$, which yields

$$\begin{aligned} P(C = c) &= \sum_m P(C = c \cap M = m) \\ &= \sum_m P(C = c \mid M = m)P(M = m) \quad (\text{by (4.1)}) \\ &= \sum_m \frac{1}{N} P(M = m) \\ &= \frac{1}{N} \quad (\text{by (4.2)}). \end{aligned}$$

Applying Equation (4.1) twice yields

$$\begin{aligned} P(M = m \mid C = c)P(C = c) &= P(C = c \cap M = m) \\ &= P(C = c \mid M = m)P(M = m). \end{aligned}$$

Since we have already proved that

$P(C = c) = 1/N = P(C = c \mid M = m)$, we can multiply by N to obtain

$$P(M = m \mid C = c) = P(M = m),$$

which says that the one-time pad has perfect secrecy.

One of the difficulties with using the one-time pad is that the number of possible keys is at least as large as the number of possible messages. Unfortunately, this is required for perfect secrecy:

Proposition

If a cryptosystem has perfect secrecy, then the number of possible keys is greater than or equal to the number of possible plaintexts.

Proof. Let M be the number of possible plaintexts and let N be the number of possible keys. Suppose $M > N$. Let c be a ciphertext. For each key k , decrypt c using the key k . This gives N possible plaintexts, and these are the only plaintexts that can encrypt to c . Since $M > N$, there is some plaintext m that is not a decryption of c .

Therefore,

$$P(M = m \mid C = c) = 0 \neq P(M = m).$$

This contradicts the assumption that the system has perfect secrecy. Therefore, $M \leq N$.

Example

Suppose a parent goes to the pet store to buy a pet for a child's birthday. The store sells 30 different pets with 3-letter names (ant, bat, cat, dog, eel, elk, ...). The parent chooses a pet at random, encrypts its name with a shift

cipher, and sends the ciphertext to let the other parent know what has been bought. The child intercepts the message, which is ZBL. The child hopes the present is a dog. Since DOG is not a shift of ZBL, the child realizes that the conditional probability

$$P(M = \text{dog} \mid C = \text{ZBL}) = 0$$

and is disappointed. Since $P(M = \text{dog}) = 1/30$ (because there are 30 equally likely possibilities), we have $P(M = \text{dog} \mid C = \text{ZBL}) \neq P(M = \text{dog})$, so there is not perfect secrecy. This is because a given ciphertext has at most 26 possible corresponding plaintexts, so knowledge of the ciphertext restricts the possibilities for the decryption. Then the child realizes that YAK is the only possible shift of ZBL, so $P(M = \text{yak} \mid C = \text{ZBL}) = 1$. This does not equal $P(M = \text{yak})$, so again we see that we don't have perfect secrecy. But now the child is happy, being the only one in the neighborhood who will have a yak as a pet.

4.5 Indistinguishability and Security

A basic requirement for a secure cryptosystem is **ciphertext indistinguishability**. This can be described by the following game:

CI Game: Alice chooses two messages m_0 and m_1 and gives them to Bob. Bob randomly chooses $b = 0$ or 1 . He encrypts m_b to get a ciphertext c , which he gives to Alice. Alice then guesses whether m_0 or m_1 was encrypted.

By randomly guessing, Alice can guess correctly about $1/2$ of the time. If there is no strategy where she guesses correctly significantly more than $1/2$ the time, then we say the cryptosystem has the ciphertext indistinguishability property.

For example, the shift cipher does not have this property. Suppose Alice chooses the two messages to be *CAT* and *DOG*. Bob randomly chooses one of them and sends back the ciphertext *PNG*. Alice observes that this cannot be a shift of *DOG* and thus concludes that Bob encrypted *CAT*.

When implemented in the most straightforward fashion, the RSA cryptosystem (see [Chapter 9](#)) also does not have the property. Since the encryption method is public, Alice can simply encrypt the two messages and compare with what Bob sends her. However, if Bob pads the messages with random bits before encryption, using a good pseudorandom number generator, then Alice should not be able to guess correctly significantly more than $1/2$ the time because she will not know the random bits used in the padding.

The one-time pad where the key is chosen randomly has ciphertext indistinguishability. Because Bob chooses b randomly,

$$P(m_0) = P(m_1) = \frac{1}{2}.$$

From the previous section, we know that

$$P(M = m_0 \mid C = c) = P(M = m_0) = \frac{1}{2}$$

and

$$P(M = m_1 \mid C = c) = P(M = m_1) = \frac{1}{2}.$$

Therefore, when Alice receives c , the two possibilities are equally likely, so the probability she guesses correctly is $1/2$.

Because the one-time pad is too unwieldy for many applications, pseudorandom generators are often used to generate substitutes for one-time pads. In [Chapter 5](#), we discuss some possibilities. For the present, we analyze how much such a choice can affect the security of the system.

A pseudorandom key generator produces N possible keys, with each possible key k having a probability $P(K = k)$. Usually, it takes an input, called a **seed**, and applies some algorithm to produce a key that “looks random.” The seed is transmitted to the decrypter of the ciphertext, who uses the seed to produce the key and then decrypt. The seed is significantly shorter than the length of the key. While the key might have, for example, 1 million bits, the seed could have only 100 bits, which makes transmission much more efficient, but this also means that there are fewer keys than with the one-time pad. Therefore, [Proposition 4.4](#) says that perfect secrecy is not possible.

If the seed had only 20 bits, it would be possible to use all of the seeds to generate all possible keys. Then, given a ciphertext and a plaintext, it would be easy to see if there is a key that encrypts the plaintext to the ciphertext. But with a seed of 100 bits, it is infeasible to list all 2^{100} seeds and find the corresponding keys. Moreover, with a good pseudorandom key generator, it should be difficult to see whether a given key is one that could be produced from some seed.

To evaluate a pseudorandom key generator, Alice (the adversary) and Bob play the following game:

R Game: *Bob flips a fair coin. If it's Heads, he chooses a number r uniformly randomly from the keyspace. If it's Tails, he chooses a pseudorandom key r . Bob sends r to Alice. Alice guesses whether r was chosen randomly or pseudorandomly.*

Of course, Alice could always guess that it's random, for example, or she could flip her own coin and use that for her guess. In these cases, her probability of guessing correctly is $1/2$. But suppose she knows something about the pseudorandom generator (maybe she has analyzed its inner workings, for example). Then she might be able to recognize sometimes that r looks like something the pseudorandom generator could produce (of course, the random generator could also produce it, but with lower probability since it has many more possible outputs). This could occasionally give Alice a slight edge in guessing. So Alice's overall probability of winning could increase slightly.

In an extreme case, suppose Alice knows that the pseudorandom number generator always has a 1 at the beginning of its output. The true random number generator will produce such an output with probability $1/2$. If Alice sees this initial 1, she guesses that the output is from the pseudorandom generator. And if this 1 is not

present, Alice knows that r is random. Therefore, Alice guesses correctly with probability $3/4$. (This is [Exercise 9](#).)

We write

$$P(\text{Alice is correct}) = \frac{1}{2} + \epsilon.$$

A good pseudorandom generator should have ϵ very small, no matter what strategy Alice uses.

But will a good pseudorandom key generator work in a one-time pad? Let's test it with the CI Game. Suppose Charles is using a pseudorandom key generator for his one-time pad and he is going to play the CI game with Alice. Moreover, suppose Alice has a strategy for winning

this CI Game with probability $\frac{1}{2} + \epsilon$ (if this happens,

then Charles's implementation is not very good). We'll show that, under these assumptions, Alice can play the R game with Bob and win with probability $\frac{1}{2} + \frac{\epsilon}{2}$.

For example, suppose that the pseudorandom number generator is such that Alice has probability at most 0.51 of winning the R Game. Then we must have $\epsilon/2 \leq .01$, so $\epsilon \leq .02$. This means that the probability that Charles wins the CI game is at most 0.52. In this way, we conclude that if the random number generator is good, then its use in a one-time pad is good.

Here is Alice's strategy for the R game. Alice and Bob do the following:

1. Bob flips a fair coin, as in the R game, and gives r to Alice. Alice wants to guess whether r is random or pseudorandom.
2. Alice uses r to play the CI game with Charles.
 1. She calls up Charles, who chooses messages m_0 and m_1 and gives them to Alice.
 2. Alice chooses $b = 0$ or 1 randomly.

3. She encrypts m_b using the key r to obtain the ciphertext
 $c = m_b \oplus r$.
 4. She sends c to Charles.
 5. Charles makes his guess for b and succeeds with
probability $\frac{1}{2} + \epsilon$.
3. Alice now uses Charles's guess to finish playing the *R* Game.
 4. If Charles guessed b correctly, her guess to Bob is that r was pseudorandom.
 5. If Charles guessed incorrectly, her guess to Bob is that r was random.

There are two ways that Alice wins the R Game. One is when Charles is correct and r is pseudorandom, and the other is when Charles is incorrect and r is random.

The probability that Charles is correct when r is pseudorandom is $\frac{1}{2} + \epsilon$, by assumption. This means that

$$\begin{aligned} P(\text{Charles is correct} \cap r \text{ is pseudorandom}) \\ = P(\text{Charles is correct} \mid r \text{ is pseudorandom}) P(r \text{ is pseudorandom}) \\ = \left(\frac{1}{2} + \epsilon\right)(1/2). \end{aligned}$$

If r is random, then Alice encrypted m_b with a true one-time pad, so Charles succeeds half the time and fails half the time. Therefore,

$$\begin{aligned} P(\text{Charles is incorrect} \cap r \text{ is random}) \\ = P(\text{Charles is incorrect} \mid r \text{ is random}) P(r \text{ is random}) \\ = \left(\frac{1}{2}\right)(1/2). \end{aligned}$$

Putting the previous two calculations together, we see that the probability that Alice wins the R Game is

$$\left(\frac{1}{2} + \epsilon\right)(1/2) + \left(\frac{1}{2}\right)(1/2) = \frac{1}{2} + \frac{1}{2}\epsilon,$$

as we claimed.

The preceding shows that if we design a good random key generator, then an adversary can gain only a very slight advantage in using a ciphertext to distinguish between two plaintexts. Unfortunately, there are not good ways to prove that a given pseudorandom number generator is good (this would require solving some major problems in complexity theory), but knowledge of where the security of the system lies is significant progress.

For a good introduction to cryptography via the language of computational security and proofs of security, see [Katz and Lindell].

4.6 Exercises

1. Alice is learning about the shift cipher. She chooses a random three-letter word (so all three-letter words in the dictionary have the same probability) and encrypts it using a shift cipher with a randomly chosen key (that is, each possible shift has probability 1/26). Eve intercepts the ciphertext mxp .

1. Compute $P(M = cat \mid C = mxp)$. (Hint: Can mxp shift to cat ?)
2. Use your result from part (a) to show that the shift cipher does not have perfect secrecy (this is also true because there are fewer keys than ciphertexts; see the proposition at the end of the first section).

2. Alice is learning more advanced techniques for the shift cipher. She now chooses a random five-letter word (so all five-letter words in the dictionary have the same probability) and encrypts it using a shift cipher with a randomly chosen key (that is, each possible shift has probability 1/26). Eve intercepts the ciphertext $evire$. Show that

$$P(M = arena \mid C = evire) = 1/2.$$

(Hint: Look at Exercise 1 in Chapter 2.)

3. Suppose a message m is chosen randomly from the set of all five-letter English words and is encrypted using an affine cipher mod 26, where the key is chosen randomly from the 312 possible keys. The ciphertext is $HHGZC$. Compute the conditional probability $\text{Prob}(m = HELLO \mid c = HHGZC)$. Use the result of this computation to determine whether or not affine ciphers have perfect secrecy.

4. Alice is learning about the Vigenère cipher. She chooses a random six-letter word (so all six-letter words in the dictionary have the same probability) and encrypts it using a Vigenère cipher with a randomly chosen key of length 3 (that is, each possible key has probability 1/26³). Eve intercepts the ciphertext $eblkfg$.

1. Compute the conditional probability $P(M = attack \mid C = eblkfg)$.
2. Use your result from part (a) to show that the Vigenère cipher does not have perfect secrecy.

5. Alice and Bob play the following game (this is the CI Game of Section 4.5). Alice chooses two two-letter words m_0 and m_1 and gives them to Bob. Bob randomly chooses $b = 0$ or 1 . He encrypts m_b using a shift cipher (with a randomly chosen shift) to get a ciphertext c , which he gives to Alice. Alice then guesses whether m_0 or m_1 was encrypted.

1. Alice chooses $m_0 = HI$ and $m_1 = NO$. What is the probability that Alice guesses correctly?
2. Give a choice of m_0 and m_1 that Alice can make so that she is guaranteed to be able to guess correctly.

6. Bob has a weak pseudorandom generator that produces N different M -bit keys, each with probability $1/N$. Alice and Bob play Game R. Alice makes a list of the N possible pseudorandom keys. If the number r that Bob gives to her is on this list, she guesses that the number is chosen pseudorandomly. If it is not on the list, she guess that it is random.

1. Show that

$$P(r \text{ is on the list} \mid r \text{ is random}) = \frac{N}{2^M}$$

and

$$P(r \text{ is on the list} \mid r \text{ is pseudorandom}) = 1.$$

2. Show that

$$\begin{aligned} P(r \text{ is random} \cap r \text{ is on the list}) &= \frac{1}{2} \left(\frac{N}{2^M} \right) \\ P(r \text{ is random} \cap r \text{ is not on the list}) &= \frac{1}{2} \left(1 - \frac{N}{2^M} \right) \\ P(r \text{ is pseudorandom} \cap r \text{ is on the list}) &= \frac{1}{2} \\ P(r \text{ is pseudorandom} \cap r \text{ is not on the list}) &= 0. \end{aligned}$$

3. Show that Alice wins with probability

$$\frac{1}{2} \left(1 - \frac{N}{2^M} \right) + \frac{1}{2}.$$

4. Show that if $N/2^M = 1 - \epsilon$ then Alice wins with probability $\frac{1}{2} + \frac{1}{2}\epsilon$. (This shows that if the pseudorandom generator misses a fraction of the possible keys, then Alice has an advantage in the game, provided that she can make a list of all possible outputs of the generator. Therefore, it is necessary to make N large enough that making such a list is infeasible.)

7. Suppose Alice knows that Bob's pseudorandom key generator has a slight bias and that with probability 51% it produces a key with more 1's than 0's. Alice and Bob play the CI Game. Alice chooses messages $m_0 = 000 \dots 0$ and $m_1 = 111 \dots 1$ to Bob, who randomly chooses $b \in \{0, 1\}$ and encrypts m_b with a one-time pad using his pseudorandom key generator. He gives the ciphertext $c = m_b \oplus r$ (where r is his pseudorandom key) to Alice. Alice computes $s = c \oplus m_0$. If s has more 1's than 0's, she guesses that $b = 0$. If not, she guesses that $b = 1$. For simplicity in the following, we assume that the message lengths are odd (so there cannot be the same number of 1's and 0's).

1. Show that exactly one of $m_0 \oplus r$ and $m_1 \oplus r$ has more 1's than 0's.
2. Show that $P(s \text{ has more 1's} \mid b = 0) = .51$
3. Show that $P(s \text{ has fewer 1's} \mid b = 1) = .51$
4. Show that Alice has a probability .51 of winning.

8. In the one-time pad, suppose that some plaintexts are more likely than others. Show that the key and the ciphertext are not independent. That is, show that there is a key k and a ciphertext c such that

$$P(C = c \cap K = k) \neq P(C = c) P(K = k).$$

(Hint: The right-hand side of this equation is independent of k and c . What about the left-hand side?)

9. Alice and Bob are playing the R Game. Suppose Alice knows that Bob's pseudorandom number generator always has a 1 at the beginning of its output. If Alice sees this initial 1, she guesses that the output is from the pseudorandom generator. And if this 1 is not present, Alice knows that r is random. Show that Alice guesses correctly with probability 3/4.
10. Suppose Bob uses a pseudorandom number generator to produce a one-time pad, but the generator has a slight bias, so each bit it produces has probability 51% of being 1 and only 49% of being 0. What strategy can Alice use so that she expects to win the CI Game more than half the time?
11. At the end of the semester, the professor randomly chooses and sends one of two possible messages:

$$m_0 = \text{YOU PASSED} \quad \text{and} \quad m_1 = \text{YOU FAILED}.$$

To add to the excitement, the professor encrypts the message using one of the following methods:

1. Shift cipher

2. Vigenère cipher with key length 3

3. One-time pad.

You receive the ciphertext and want to decide whether the professor sent m_0 or m_1 . For each method (a), (b), (c), explain how to decide which message was sent or explain why it is impossible to decide. You may assume that you know which method is being used. (For the Vigenère, do not do frequency analysis; the message is too short.)

12. On Groundhog Day, the groundhog randomly chooses and sends one of two possible messages:

$$m_0 = \text{SIXMOREWEEKSOFWINTER}$$

$$m_1 = \text{SPRINGARRIVESINMARCH}.$$

To add to the mystery, the groundhog encrypts the message using one of the following methods: shift cipher, Vigenère cipher with key length 4 (using four distinct shifts), one-time pad. For each of the following ciphertexts, determine which encryption methods could have produced that ciphertext, and for each of these possible encryption methods, decrypt the message or explain why it is impossible to decrypt.

1. ABCDEFGHIJKLMNOPQRST

2. UKZOQQTGYGGMUQHYKPVGT

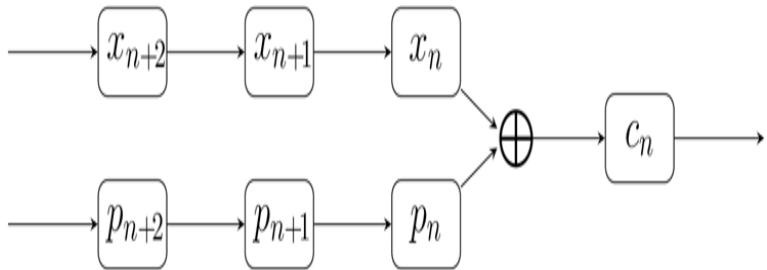
3. UQUMPHDVTJYIUJQQCSFL.

13. Alice encrypts the messages M_1 and M_2 with the same one-time pad using only capital letters and spaces, as in [Section 4.3](#). Eve knows this, intercepts the ciphertexts C_1 and C_2 , and also learns that the decryption of M_1 is *THE LETTER * ON THE MAP GIVES THE LOCATION OF THE TREASURE*. Unfortunately for Eve, she cannot read the missing letter *. However, the 12th group of seven bits in C_1 is 1001101 and the 12th group in C_2 is 0110101. Find the missing letter.

Chapter 5 Stream Ciphers

The one-time pad provides a strong form of secrecy, but since key transmission is difficult, it is desirable to devise substitutes that are easier to use. Stream ciphers are one way of achieving this goal. As in the one-time pad, the plaintext is written as a string of bits. Then a binary keystream is generated and XORed with the plaintext to produce the ciphertext.

Figure 5.1 Stream cipher encryption



p_n = plaintext bit, x_n = key bit, c_n = ciphertext bit.

Figure 5.1 Full Alternative Text

For the system to be secure, the keystream needs to approximate a random sequence, so we need a good source of random-looking bits. In [Section 5.1](#), we discuss pseudorandom number generators. In [Sections 5.2](#) and [5.3](#), we describe two commonly used stream ciphers and the pseudorandom number generators that they use. Although they have security weaknesses, they give an idea of methods that can be used.

In the next chapter, we discuss block ciphers and various modes of operations. Some of the most secure stream ciphers are actually good block ciphers used, for

example, in OFB or CTR mode. See Subsections [6.3.4](#) and [6.3.5](#).

There is one problem that is common to all stream ciphers that are obtained by XORing pseudorandom numbers with plaintext and is one of the reasons that authentication and message integrity checks are added to protect communications. Suppose Eve knows where the word “good” occurs in a plaintext that has been encrypted with a stream cipher. If she intercepts the ciphertext, she can XOR the bits for good \oplus evil at the appropriate place in the ciphertext before continuing the transmission of the ciphertext. When the ciphertext is decrypted, “good” will be changed to “evil.” This type of attack was one of the weaknesses of the WEP system, which is discussed in [Section 14.3](#).

5.1 Pseudorandom Bit Generation

The one-time pad and many other cryptographic applications require sequences of random bits. Before we can use a cryptographic algorithm, such as DES ([Chapter 7](#)) or AES ([Chapter 8](#)), it is necessary to generate a sequence of random bits to use as the key.

One way to generate random bits is to use natural randomness that occurs in nature. For example, the thermal noise from a semiconductor resistor is known to be a good source of randomness. However, just as flipping coins to produce random bits would not be practical for cryptographic applications, most natural conditions are not practical due to the inherent slowness in sampling the process and the difficulty of ensuring that an adversary does not observe the process. We would therefore like a method for generating randomness that can be done in software. Most computers have a method for generating random numbers that is readily available to the user. For example, the standard C library contains a function `rand()` that generates pseudorandom numbers between 0 and 65535. This pseudorandom function takes a **seed** as input and produces an output bitstream.

The `rand()` function and many other pseudorandom number generators are based on linear congruential generators. A **linear congruential generator** produces a sequence of numbers x_1, x_2, \dots , where

$$x_n = ax_{n-1} + b \pmod{m}.$$

The number x_0 is the initial seed, while the numbers a, b , and m are parameters that govern the relationship. The

use of pseudorandom number generators based on linear congruential generators is suitable for experimental purposes, but is highly discouraged for cryptographic purposes. This is because they are predictable (even if the parameters a , b , and m are not known), in the sense that an eavesdropper can use knowledge of some bits to predict future bits with fairly high probability. In fact, it has been shown that any polynomial congruential generator is cryptographically insecure.

In cryptographic applications, we need a source of bits that is nonpredictable. We now discuss two ways to create such nonpredictable bits.

The first method uses one-way functions. These are functions $f(x)$ that are easy to compute but for which, given y , it is computationally infeasible to solve $y = f(x)$ for x . Suppose that we have such a one-way function f and a random seed s . Define $x_j = f(s + j)$ for $j = 1, 2, 3, \dots$. If we let b_j be the least significant bit of x_j , then the sequence b_0, b_1, \dots will often be a pseudorandom sequence of bits (but see [Exercise 14](#)). This method of random bit generation is often used, and has proven to be very practical. Two popular choices for the one-way function are DES ([Chapter 7](#)) and SHA, the Secure Hash Algorithm ([Chapter 11](#)). As an example, the cryptographic pseudorandom number generator in the OpenSSL toolkit (used for secure communications over the Internet) is based on SHA.

Another method for generating random bits is to use an intractable problem from number theory. One of the most popular cryptographically secure pseudorandom number generators is the **Blum-Blum-Shub (BBS) pseudorandom bit generator**, also known as the quadratic residue generator. In this scheme, one first generates two large primes p and q that are both congruent to $3 \pmod{4}$. We set $n = pq$ and choose a random integer x that is relatively prime to n . To

initialize the BBS generator, set the initial seed to $x_0 \equiv x^2 \pmod{n}$. The BBS generator produces a sequence of random bits b_1, b_2, \dots by

1. $x_j \equiv x_{j-1}^2 \pmod{n}$
2. b_j is the least significant bit of x_j .

Example

Let

$$p = 24672462467892469787 \text{ and } q = 396736894567834589803,$$

$$n = 9788476140853110794168855217413715781961.$$

Take $x = 873245647888478349013$. The initial seed is

$$\begin{aligned} x_0 &\equiv x^2 \pmod{n} \\ &\equiv 8845298710478780097089917746010122863172. \end{aligned}$$

The values for x_1, x_2, \dots, x_8 are

$$\begin{aligned} x_1 &\equiv 7118894281131329522745962455498123822408 \\ x_2 &\equiv 314517460888893164151380152060704518227 \\ x_3 &\equiv 4898007782307156233272233185574899430355 \\ x_4 &\equiv 3935457818935112922347093546189672310389 \\ x_5 &\equiv 675099511510097048901761303198740246040 \\ x_6 &\equiv 4289914828771740133546190658266515171326 \\ x_7 &\equiv 4431066711454378260890386385593817521668 \\ x_8 &\equiv 7336876124195046397414235333675005372436. \end{aligned}$$

Taking the least significant bit of each of these, which is easily done by checking whether the number is odd or even, produces the sequence

$$b_1, \dots, b_8 = 0, 1, 1, 1, 0, 0, 0, 0.$$

The Blum-Blum-Shub generator is very likely unpredictable. See [Blum-Blum-Shub]. A problem with BBS is that it can be slow to calculate. One way to improve its speed is to extract the k least significant bits

of x_j . As long as $k \leq \log_2 \log_2 n$, , this seems to be cryptographically secure.

5.2 Linear Feedback Shift Register Sequences

Note: In this section, all congruences are mod 2.

In many situations involving encryption, there is a trade-off between speed and security. If one wants a very high level of security, speed is often sacrificed, and vice versa. For example, in cable television, many bits of data are being transmitted, so speed of encryption is important. On the other hand, security is not usually as important since there is rarely an economic advantage to mounting an expensive attack on the system.

In this section, we describe a method that could be used when speed is more important than security. However, the real use is as one building block in more complex systems.

The sequence

01000010010110011111000110111010100001001011001111

can be described by giving the initial values

$$x_1 \equiv 0, x_2 \equiv 1, x_3 \equiv 0, x_4 \equiv 0, x_5 \equiv 0$$

and the linear recurrence relation

$$x_{n+5} \equiv x_n + x_{n+2} \pmod{2}.$$

This sequence repeats after 31 terms.

For another example, see [Example 18](#) in the Computer Appendices.

More generally, consider a linear recurrence relation of **length m** :

$$x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \cdots + c_{m-1} x_{n+m-1} \pmod{2},$$

where the coefficients c_0, c_1, \dots are integers. If we specify the **initial values**

$$x_1, x_2, \dots, x_m,$$

then all subsequent values of x_n can be computed using the recurrence. The resulting sequence of 0s and 1s can be used as the key for encryption. Namely, write the plaintext as a sequence of 0s and 1s, then add an appropriate number of bits of the key sequence to the plaintext mod 2, bit by bit. For example, if the plaintext is 1011001110001111 and the key sequence is the example given previously, we have

(plaintext)	1011001110001111
(key) \oplus	0100001001011001
(ciphertext)	1111000111010110

Decryption is accomplished by adding the key sequence to the ciphertext in exactly the same way.

One advantage of this method is that a key with large period can be generated using very little information. The long period gives an improvement over the Vigenère method, where a short period allowed us to find the key. In the above example, specifying the initial vector $\{0, 1, 0, 0, 0\}$ and the coefficients $\{1, 0, 1, 0, 0\}$ yielded a sequence of period 31, so 10 bits were used to produce 31 bits. It can be shown that the recurrence

$$x_{n+31} \equiv x_n + x_{n+3}$$

and any nonzero initial vector will produce a sequence with period $2^{31} - 1 = 2147483647$. Therefore, 62 bits produce more than two billion bits of key. This is a great advantage over a one-time pad, where the full two billion bits must be sent in advance.

This method can be implemented very easily in hardware using what is known as a **linear feedback shift register** (LFSR) and is very fast. In Figure 5.2 we depict

an example of a linear feedback shift register in a simple case. More complicated recurrences are implemented using more registers and more XORs.

Figure 5.2 A Linear Feedback Shift Register Satisfying

$$x_{n+3} = x_{n+1} + x_n.$$

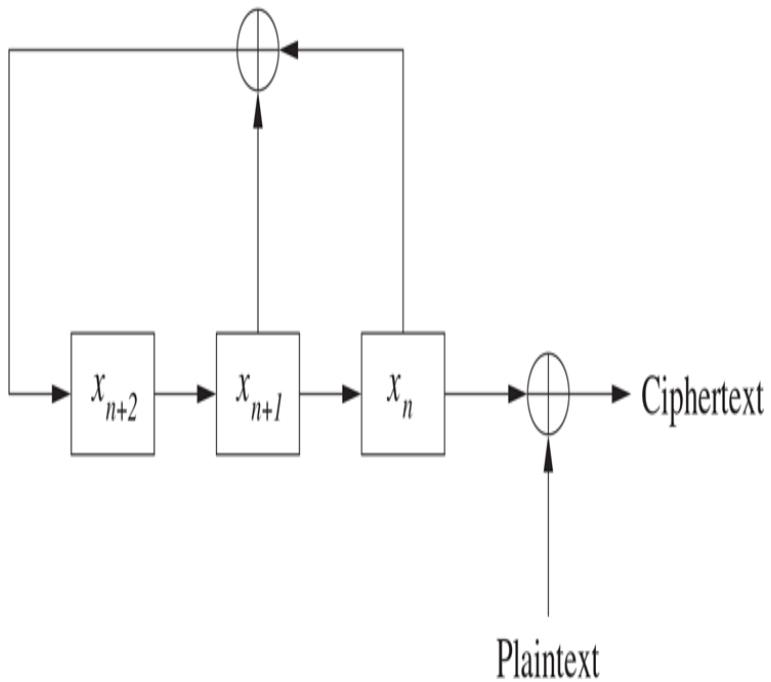


Figure 5.2 Full Alternative Text

For each increment of a counter, the bit in each box is shifted to other boxes as indicated, with \oplus denoting the addition mod 2 of the incoming bits. The output, which is the bit x_n , is added to the next bit of plaintext to produce the ciphertext. The diagram in Figure 5.2 represents the recurrence $x_{n+3} \equiv x_{n+1} + x_n$. Once the initial values x_1, x_2, x_3 are specified, the machine produces the subsequent bits very efficiently.

Unfortunately, the preceding encryption method succumbs easily to a known plaintext attack. More precisely, if we know only a few consecutive bits of plaintext, along with the corresponding bits of ciphertext, we can determine the recurrence relation and therefore compute all subsequent bits of the key. By subtracting (or adding; it's all the same mod 2) the plaintext from the ciphertext mod 2, we obtain the bits of the key. Therefore, for the rest of this discussion, we will ignore the ciphertext and plaintext and assume we have discovered a portion of the key sequence. Our goal is to use this portion of the key to deduce the coefficients of the recurrence and consequently compute the rest of the key.

For example, suppose we know the initial segment 011010111100 of the sequence 0110101111000100110101111 . . . , which has period 15, and suppose we know it is generated by a linear recurrence. How do we determine the coefficients of the recurrence? We do not necessarily know even the length, so we start with length 2 (length 1 would produce a constant sequence). Suppose the recurrence is $x_{n+2} = c_0 x_n + c_1 x_{n+1}$. Let $n = 1$ and $n = 2$ and use the known values $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$. We obtain the equations

$$\begin{aligned} 1 &\equiv c_0 \cdot 0 + c_1 \cdot 1 & (n = 1) \\ 0 &\equiv c_0 \cdot 1 + c_1 \cdot 1 & (n = 2). \end{aligned}$$

In matrix form, this is

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The solution is $c_0 = 1$, $c_1 = 1$, so we guess that the recurrence is $x_{n+2} \equiv x_n + x_{n+1}$. Unfortunately, this is not correct since $x_6 \neq x_4 + x_5$. Therefore, we try length 3. The resulting matrix equation is

$$\begin{array}{ccc} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{array} \quad \begin{array}{c} c_0 \\ c_1 \\ c_2 \end{array} \equiv \begin{array}{c} 0 \\ 1 \\ 0 \end{array} .$$

The determinant of the matrix is 0 mod 2; in fact, the equation has no solution. We can see this because every column in the matrix sums to 0 mod 2, while the vector on the right does not.

Now consider length 4. The matrix equation is

$$\begin{array}{cccc} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{array} \quad \begin{array}{c} c_0 \\ c_1 \\ c_2 \\ c_3 \end{array} \equiv \begin{array}{c} 1 \\ 0 \\ 1 \\ 1 \end{array} .$$

The solution is $c_0 = 1, c_1 = 1, c_2 = 0, c_3 = 0$. The resulting recurrence is now conjectured to be

$$x_{n+4} \equiv x_n + x_{n+1}.$$

A quick calculation shows that this generates the remaining bits of the piece of key that we already know, so it is our best guess for the recurrence that generates the key sequence.

What happens if we try length 5? The matrix equation is

$$\begin{array}{cccccc} 0 & 1 & 1 & 0 & 1 & c_0 & 0 \\ 1 & 1 & 0 & 1 & 0 & c_1 & 1 \\ 1 & 0 & 1 & 0 & 1 & c_2 & \equiv 1 \\ 0 & 1 & 0 & 1 & 1 & c_3 & 1 \\ 1 & 0 & 1 & 1 & 1 & c_4 & 1 \end{array} .$$

The determinant of the matrix is 0 mod 2. Why? Notice that the last row is the sum of the first and second rows. This is a consequence of the recurrence relation: $x_5 \equiv x_1 + x_2, x_6 \equiv x_2 + x_3$, etc. As in linear algebra with real or complex numbers, if one row of a matrix is a linear combination of other rows, then the determinant is 0.

Similarly, if we look at the 6×6 matrix, we see that the 5th row is the sum of the first and second rows, and the

6th row is the sum of the second and third rows, so the determinant is 0 mod 2. In general, when the size of the matrix is larger than the length of the recurrence relation, the relation forces one row to be a linear combination of other rows, hence the determinant is 0 mod 2.

The general situation is as follows. To test for a recurrence of length m , we assume we know x_1, x_2, \dots, x_{2m} . The matrix equation is

$$\begin{array}{cccccc} x_1 & x_2 & \cdots & x_m & c_0 & x_{m+1} \\ x_2 & x_3 & \cdots & x_{m+1} & c_1 & x_{m+2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ x_m & x_{m+1} & \cdots & x_{2m-1} & c_{m-1} & x_{2m} \end{array} \equiv \cdot$$

We show later that the matrix is invertible mod 2 if and only if there is no linear recurrence of length less than m that is satisfied by $x_1, x_2, \dots, x_{2m-1}$.

A strategy for finding the coefficients of the recurrence is now clear. Suppose we know the first 100 bits of the key. For $m = 2, 3, 4, \dots$, form the $m \times m$ matrix as before and compute its determinant. If several consecutive values of m yield 0 determinants, stop. The last m to yield a nonzero (i.e., 1 mod 2) determinant is probably the length of the recurrence. Solve the matrix equation to get the coefficients c_0, \dots, c_{m-1} . It can then be checked whether the sequence that this recurrence generates matches the sequence of known bits of the key. If not, try larger values of m .

Suppose we don't know the first 100 bits, but rather some other 100 consecutive bits of the key. The same procedure applies, using these bits as the starting point. In fact, once we find the recurrence, we can also work backwards to find the bits preceding the starting point.

Here is an example. Suppose we have the following sequence of 100 bits:

10011001001110001100010100011110110011111010101001
 01101101011000011011100101011110000000100010010000.

The first 20 determinants, starting with $m = 1$, are

$$1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.$$

A reasonable guess is that $m = 8$ gives the last nonzero determinant. When we solve the matrix equation for the coefficients we get

$$\{c_0, c_1, \dots, c_7\} = \{1, 1, 0, 0, 1, 0, 0, 0\},$$

so we guess that the recurrence is

$$x_{n+8} \equiv x_n + x_{n+1} + x_{n+4}.$$

This recurrence generates all 100 terms of the original sequence, so we have the correct answer, at least based on the knowledge that we have.

Suppose that the 100 bits were in the middle of some sequence, and we want to know the preceding bits. For example, suppose the sequence starts with x_{17} , so $x_{17} = 1, x_{18} = 0, x_{19} = 0, \dots$. Write the recurrence as

$$x_n \equiv x_{n+1} + x_{n+4} + x_{n+8}$$

(it might appear that we made some sign errors, but recall that we are working mod 2, so $-x_n \equiv x_n$ and $-x_{n+8} \equiv x_{n+8}$). Letting $n = 16$ yields

$$\begin{aligned} x_{16} &\equiv x_{17} + x_{20} + x_{24} \\ &\equiv 1 + 0 + 1 \equiv 0. \end{aligned}$$

Continuing in this way, we successively determine $x_{15}, x_{14}, \dots, x_1$.

For more examples, see Examples 19 and 20 in the Computer Appendices.

We now prove the result we promised.

Proposition

Let x_1, x_2, x_3, \dots be a sequence of bits produced by a linear recurrence mod 2. For each $n \geq 1$, let

$$M_n = \begin{matrix} x_1 & x_2 & \cdots & x_n \\ x_2 & x_3 & \cdots & x_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_{n+1} & \cdots & x_{2n-1} \end{matrix}.$$

Let N be the length of the shortest recurrence that generates the sequence x_1, x_2, x_3, \dots . Then $\det(M_N) \equiv 1 \pmod{2}$ and $\det(M_n) \equiv 0 \pmod{2}$ for all $n > N$.

Proof. We first make a few remarks on the length of recurrences. A sequence could satisfy a length 3 relation such as $x_{n+3} \equiv x_{n+2}$. It would clearly then also satisfy shorter relations such as $x_{n+1} = x_n$ (at least for $n \geq 2$). However, there are less obvious ways that a sequence could satisfy a recurrence of length less than expected. For example, consider the relation

$x_{n+4} \equiv x_{n+3} + x_{n+1} + x_n$. Suppose the initial values of the sequence are 1, 1, 0, 1. The recurrence allows us to compute subsequent terms:

1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1. . . . It is easy to see that the sequence satisfies $x_{n+2} \equiv x_{n+1} + x_n$.

If there is a recurrence of length N and if $n > N$, then one row of the matrix M_n is congruent mod 2 to a linear combination of other rows. For example, if the recurrence is $x_{n+3} = x_{n+2} + x_n$, then the fourth row is the sum of the first and third rows. Therefore, $\det(M_n) \equiv 0 \pmod{2}$ for all $n > N$.

Now suppose $\det(M_N) \equiv 0 \pmod{2}$. Then there is a nonzero row vector $b = (b_0, \dots, b_{N-1})$ such that $bM_N \equiv 0$. We'll show that this gives a recurrence relation for the sequence x_1, x_2, x_3, \dots and that the length of this relation is less than N . This contradicts the

assumption that N is smallest. This contradiction implies that $\det(M_N) \equiv 1 \pmod{2}$.

Let the recurrence of length N be

$$x_{n+N} \equiv c_0 x_n + \cdots + c_{N-1} x_{n+N-1}.$$

For each $i \geq 0$, let

$$M^{(i)} = \begin{matrix} & x_{i+1} & x_{i+2} & \cdots & x_{i+N} \\ & x_{i+2} & x_{i+3} & \cdots & x_{i+N+1} \\ & \vdots & \vdots & \ddots & \vdots \\ & x_{i+N} & x_{i+N+1} & \cdots & x_{i+2N-1} \end{matrix}.$$

Then $M^{(0)} = M_N$. The recurrence relation implies that

$$M^{(i)} \begin{matrix} c_0 & x_{i+N+1} \\ c_1 & x_{i+N+2} \\ \vdots & \vdots \\ c_{N-1} & x_{i+2N} \end{matrix},$$

which is the last column of $M^{(i+1)}$.

By the choice of b , we have $bM^{(0)} = bM_N = 0$.

Suppose that we know that $bM^{(i)} = 0$ for some i . Then

$$b \begin{matrix} x_{i+N+1} \\ x_{i+N+2} \\ \vdots \\ x_{i+2N} \end{matrix} \equiv bM^{(i)} \begin{matrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{matrix} \equiv 0.$$

Therefore, b annihilates the last column of $M^{(i+1)}$. Since the remaining columns of $M^{(i+1)}$ are columns of $M^{(i)}$, we find that $bM^{(i+1)} \equiv 0$. By induction, we obtain $bM^{(i)} \equiv 0$ for all $i \geq 0$.

Let $n \geq 1$. The first column of $M^{(n-1)}$ yields

$$b_0 x_n + b_1 x_{n+1} + \cdots + b_{N-1} x_{n+N-1} \equiv 0.$$

Since b is not the zero vector, $b_j \neq 0$ for at least one j .

Let m be the largest j such that $b_j \neq 0$, which means that $b_m = 1$. We are working mod 2, so

$b_m x_{n+m} \equiv -x_{n+m}$. Therefore, we can rearrange the relation to obtain

$$x_{n+m} \equiv b_0 x_n + b_1 x_{n+1} + \cdots + b_{m-1} x_{n+m-1}.$$

This is a recurrence of length m . Since $m \leq N - 1$, and N is assumed to be the shortest possible length, we have a contradiction. Therefore, the assumption that $\det(M_N) \equiv 0$ must be false, so $\det(M_N) \equiv 1$. This completes the proof.

Finally, we make a few comments about the period of a sequence. Suppose the length of the recurrence is m . Any m consecutive terms of the sequence determine all future elements, and, by reversing the recurrence, all previous values, too. Clearly, if we have m consecutive 0s, then all future values are 0. Also, all previous values are 0. Therefore, we exclude this case from consideration. There are $2^m - 1$ strings of 0s and 1s of length m in which at least one term is nonzero. Therefore, as soon as there are more than $2^m - 1$ terms, some string of length m must occur twice, so the sequence repeats. The period of the sequence is at most $2^m - 1$.

Associated to a recurrence

$x_{n+m} \equiv c_0 x_n + c_1 x_{n+1} + \cdots + c_{m-1} x_{n+m-1} \pmod{2}$, there is a polynomial

$$f(T) = T^m - c_{m-1} T^{m-1} - \cdots - c_0.$$

If $f(T)$ is irreducible mod 2 (this means that it is not congruent to the product of two lower-degree polynomials), then it can be shown that the period divides $2^m - 1$. An interesting case is when $2^m - 1$ is prime (these are called Mersenne primes). If the period isn't 1, that is, if the sequence is not constant, then the period in this special case must be maximal, namely $2^m - 1$ (see [Section 3.11](#)). The example where the period is $2^{31} - 1$ is of this type.

Linear feedback shift register sequences have been studied extensively. For example, see [Golomb] or [van der Lubbe].

One way of thwarting the above attack is to use nonlinear recurrences, for example,

$$x_{n+3} \equiv x_{n+2}x_n + x_{n+1}.$$

Moreover, a look-up table that takes inputs x_n, x_{n+1}, x_{n+2} and outputs a bit x_{n+3} could be used, or several LFSRs could be combined nonlinearly and some of these LFSRs could have irregular clocking. Generally, these systems are somewhat harder to break. However, we shall not discuss them here.

5.3 RC4

RC4 is a stream cipher that was developed by Rivest and has been widely used because of its speed and simplicity. The algorithm was originally secret, but it was leaked to the Internet in 1994 and has since been extensively analyzed. In particular, certain statistical biases were found in the keystream it generated, especially in the initial bits. Therefore, often a version called RC4-drop[n] is used, in which the first n bits are dropped before starting the keystream. However, this version is still not recommended for situations requiring high security.

To start the generation of the keystream for RC4, the user chooses a key, which is a binary string between 40 and 256 bits long. This is put into the Key Scheduling Algorithm. It starts with an array S consisting of the numbers from 0 to 255, regarded as 8-bit bytes, and outputs a permutation of these entries, as follows:

Algorithm 1 RC4 Key Scheduling Algorithm

- 1: **for** i from 0 to 255 **do**
- 2: $S[i] := i$
- 3: $j := 0$
- 4: **for** i from 0 to 255 **do**
- 5: $j := (j + S[i] + \text{key}[i \bmod \text{keylength}]) \bmod 256$
- 6: $\text{swap}(S[i], S[j])$

This algorithm starts by initializing the entries in S as $S[i] = i$ for i running from 0 through 255. Suppose the

user-supplied key is $10100000 \dots$ (let's say the key length is 40).

The algorithm starts with $j = 0$ and $i = 0$. The value of j is updated to

$j + S[i] + \text{key}[i \bmod 40] = 0 + 0 + 1 = 1$. Then $S[i] = S[0] = 0$ and $S[j] = S[1] = 1$ are swapped, so now $S[0] = 1$ and $S[1] = 0$.

We now move to $i = 1$. The value $j = 1$ is updated to $1 + 0 + \text{key}[1] = 1$, so $S[1]$ is swapped with itself, which means it is not changed here.

We now move to $i = 2$. The value $j = 1$ is updated to $1 + 2 + \text{key}[2] = 4$, so $S[2] = 2$ is swapped with $S[4] = 4$, yielding $S[2] = 4$ and $S[4] = 2$.

We now move to $i = 3$. The value $j = 4$ is updated to $4 + 3 + 0 = 7$, so $S[3]$ and $S[7]$ are swapped, yielding $S[3] = 7$ and $S[7] = 3$.

Let's look at one more value of i , namely, $i = 4$. The value $j = 7$ is updated to $7 + 2 + 0 = 9$ (recall that $S[4]$ became 2 earlier), and we obtain $S[4] = 9$ and $S[9] = 2$.

This process continues through $i = 255$ and yields an array S of length 256 consisting of a permutation of the numbers from 0 through 255.

The array S is entered into the Pseudorandom Generation Algorithm.

Algorithm 2 RC4 Pseudorandom Generation Algorithm (PRGA)

- $i := 0$
- $j := 0$
- **while** GeneratingOutput: **do**
 - $i := (i + 1) \bmod 256$
 - $j := (j + S[i]) \bmod 256$
 - $\text{swap}(S[i], S[j])$
 - output $S[(S[i] + S[j]) \bmod 256]$

This algorithm runs as long as needed and each round outputs a number between 0 and 255, regarded as an 8-bit byte. This byte is XORed with the corresponding byte of the plaintext to yield the ciphertext.

Weaknesses. Generally, the keystream that is output by a stream cipher should be difficult to distinguish from a randomly generated bitstream. For example, the R Game (see [Section 4.5](#)) could be played, and the probability of winning should be negligibly larger than $1/2$. For RC4, there are certain observable biases. The second byte in the output should be 0 with probability $1/256$. However, Mantin and Shamir [Mantin-Shamir] showed that this byte is 0 with twice that probability. Moreover, they found that the probability that the first two bytes are simultaneously 0 is $3/256^2$ instead of the expected $1/256^2$.

Biases have also been found in the state S that is output by the Key Scheduling Algorithm. For example, the probability that $S[0] = 1$ is about 37% larger than the expected probability of $1/256$, while the probability that $S[0] = 255$ is 26% less than expected.

Although any key length from 40 to 255 bits can be chosen, the use of small key sizes is not recommended because the algorithm can succumb to a brute force attack.

5.4 Exercises

1. A sequence generated by a length 3 recurrence starts 001110. Find the next four elements of the sequence.
2. The LFSR sequence 10011101 \dots is generated by a recurrence relation of length 3:
 $x_{n+3} \equiv c_0x_n + c_1x_{n+1} + c_2x_{n+2} \pmod{2}$. Find the coefficients c_0, c_1, c_2 .
3. The LFSR sequence 100100011110 \dots is generated by a recurrence relation of length 4:
 $x_{n+4} \equiv c_0x_n + c_1x_{n+1} + c_2x_{n+2} + c_3x_{n+3} \pmod{2}$. Find the coefficients c_0, c_1, c_2, c_3 .
4. The LFSR sequence 10111001 \dots is generated by a recurrence of length 3: $x_{n+3} \equiv c_0x_n + c_1x_{n+1} + c_2x_{n+2} \pmod{2}$. Find the coefficients c_0, c_1 , and c_2 .

5. Suppose we build an LFSR machine that works mod 3 instead of mod 2. It uses a recurrence of length 2 of the form

$$x_{n+2} \equiv c_0x_n + c_1x_{n+1} \pmod{3}$$

to generate the sequence 1, 1, 0, 2, 2, 0, 1, 1. Set up and solve the matrix equation to find the coefficients c_0 and c_1 .

6. The sequence $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1, x_5 = 0, x_6 = 0, x_7 = 1, x_8 = 0, x_9 = 1, \dots$ is generated by a recurrence relation

$$x_{n+3} \equiv c_0x_n + c_1x_{n+1} + c_2x_{n+2} \pmod{2}.$$

Determine x_{10}, x_{11}, x_{12} .

7. Consider the sequence starting $k_1 = 1, k_2 = 0, k_3 = 1$ and defined by the length 3 recurrence $k_{n+3} \equiv k_n + k_{n+1} + k_{n+2} \pmod{2}$. This sequence can also be given by a length 2 recurrence. Determine this length 2 recurrence by setting up and solving the appropriate matrix equations.

8. Suppose we build an LFSR-type machine that works mod 2. It uses a recurrence of length 2 of the form

$$x_{n+2} \equiv c_0x_n + c_1x_{n+1} + 1 \pmod{2}$$

to generate the sequence 1,1,0,0,1,1,0,0. Find c_0 and c_1 .

9. Suppose you modify the LFSR method to work mod 5 and you use a (not quite linear) recurrence relation

$$x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} + 2 \pmod{5},$$

$$x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0.$$

Find the coefficients c_0 and c_1 .

10. 1. Suppose you make a modified LFSR machine using the recurrence relation
 $x_{n+2} \equiv A + Bx_n + Cx_{n+1} \pmod{2}$, where A, B, C are constants. Suppose the output starts 0, 1, 1, 0, 0, 1, 1, 0, 0. Find the constants A, B, C .
2. Show that the sequence does not satisfy any linear recurrence of the form
 $x_{n+2} \equiv c_0 x_n + c_1 x_{n+1} \pmod{2}$.
11. Bob has a great idea to generate pseudorandom bytes. He takes the decimal expansion of π , which we assume is random, and chooses three consecutive digits in this expansion, starting at a randomly chosen point. He then regards what he gets as a three-digit integer, which he writes as a 10-digit number in binary. Finally, he chooses the last eight binary digits to get a byte. For example, if the digits of π that he chooses are 159, he changes this to 001001111. This yields the byte 1001111.
1. Show that this pseudorandom number generator produces some bytes more often than others.
2. Suppose Bob modifies his algorithm so that if his three-digit decimal integer is greater than or equal to 512, then he discards it and tries again. Show that this produces random output (assuming the same is true for π).
12. Suppose you have a Geiger counter and a radioactive source. If the Geiger counter counts an even number of radioactive particles in a second, you write 0. If it records an odd number of particles, you write 1. After a period of time, you have a binary sequence. It is reasonable to expect that the probability p_n that n particles are counted in a second satisfies a Poisson distribution
- $$p_n = e^{-\lambda} \frac{\lambda^n}{n!} \text{ for } n \geq 0,$$
- where λ is a parameter (in fact, λ is the average number of particles per second).
1. Show that if $0 < \lambda < 1$ then $p_0 > p_1 > p_2 \dots$
2. Show that if $\lambda < 1$ then the binary sequence you obtain is expected to have more 0s than 1s.
3. More generally, show that, whenever $\lambda \geq 0$,

$$\text{Prob}(n \text{ is even}) = e^{-\lambda} \cosh(\lambda)$$

$$\text{Prob}(n \text{ is odd}) = e^{-\lambda} \sinh(\lambda).$$

4. Show that for every $\lambda \geq 0$,

$$\text{Prob}(n \text{ is even}) > \text{Prob}(n \text{ is odd}).$$

This problem shows that, although a Geiger counter might be a good source of randomness, the naive method of using it to obtain a pseudorandom sequence is biased.

13.
 1. Suppose that during the PRGA of RC4, there occur values of $i = i_0$ and $j = j_0$ such that $j_0 = i_0 + 1$ and $S[i_0 + 1] = 1$. The next values of i, j in the algorithm are i_1, j_1 , with $i_1 = i + 1$ and $j_1 = j + 1$. Show that $S[i_1 + 1] = 1$, so this property continues for all future i, j .
 2. The values of i, j before i_0, j_0 are $i^- = i_0 - 1$ and $j^- = j_0 - 1$. Show that $S[i^-] = 1$, so if this property occurs, then it occurred for all previous values of i, j .
 3. The starting values of i and j are $i = 0$ and $j = 0$. Use this to show that there are never values of i, j such that $j = i + 1$ and $S[i] = 1$.
14. Let $f(x)$ be a one-way function. In [Section 5.1](#), it was pointed out that usually the least significant bits of $f(s + j)$ for $j = 1, 2, 3, \dots$ (s is a seed) can be used to give a pseudorandom sequence of bits. Show how to append some bits to $f(x)$ to obtain a new one-way function for which the sequence of least significant bits is not pseudorandom.

5.5 Computer Problems

1. The following sequence was generated by a linear feedback shift register. Determine the recurrence that generated it.

```
1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1,
0, 0, 1, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1,
1, 1, 1, 1, 0,
0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
1, 1, 1, 0, 1,
1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0,
1, 1, 0, 0, 0,
1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 1, 0, 1,
1, 1, 1, 1, 1
```

(It is stored in the downloadable computer files
(bit.ly/2JbcS6p) under the name *L101*.)

2. The following are the first 100 terms of an LFSR output. Find the coefficients of the recurrence.
-

```
1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0,
0, 0, 1, 1, 0,
0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1,
1, 0, 0, 1, 1,
1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1,
1, 0, 1, 1, 0,
1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1,
0, 0, 1, 0, 1,
0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 1, 0, 0,
1, 0, 0, 0, 0
```

(It is stored in the downloadable computer files
(bit.ly/2JbcS6p) under the name *L100*.)

3. The following ciphertext was obtained by XORing an LFSR output with the plaintext.
-

```
0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0,
```

0, 1, 1, 1, 0,
1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0,
0, 1, 0, 1, 0,
1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1

Suppose you know the plaintext starts

1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
0

Find the plaintext. (The ciphertext is stored in the downloadable computer files (bit.ly/2JbcS6p) under the name *L011*.)

Chapter 6 Block Ciphers

6.1 Block Ciphers

In many classical cryptosystems, changing one letter in the plaintext changes exactly one letter in the ciphertext. In the shift, affine, and substitution ciphers, a given letter in the ciphertext always comes from exactly one letter in the plaintext. This greatly facilitates finding the key using frequency analysis. In the Vigenère system, the use of blocks of letters, corresponding to the length of the key, makes the frequency analysis more difficult, but still possible, since there is no interaction among the various letters in each block. Block ciphers avoid these problems by encrypting blocks of several letters or numbers simultaneously. A change of one character in a plaintext block should change potentially all the characters in the corresponding ciphertext block.

The Playfair cipher in [Section 2.6](#) is a simple example of a block cipher, since it takes two-letter blocks and encrypts them to two-letter blocks. A change of one letter of a plaintext pair always changes at least one letter, and usually both letters, of the ciphertext pair. However, blocks of two letters are too small to be secure, and frequency analysis, for example, is usually successful.

Many of the modern cryptosystems that will be treated later in this book are block ciphers. For example, DES operates on blocks of 64 bits. AES uses blocks of 128 bits. RSA sometimes uses blocks more than 1000 bits long, depending on the modulus used. All of these block lengths are long enough to be secure against attacks such as frequency analysis.

Claude Shannon, in one of the fundamental papers on the theoretical foundations of cryptography [Shannon1], gave two properties that a good cryptosystem should have in order to hinder statistical analysis: **diffusion** and **confusion**.

Diffusion means that if we change a character of the plaintext, then several characters of the ciphertext should change, and, similarly, if we change a character of the ciphertext, then several characters of the plaintext should change. This means that frequency statistics of letters, digrams, etc. in the plaintext are diffused over several characters in the ciphertext, which means that much more ciphertext is needed to do a meaningful statistical attack.

Confusion means that the key does not relate in a simple way to the ciphertext. In particular, each character of the ciphertext should depend on several parts of the key. When a situation like this happens, the cryptanalyst probably needs to solve for the entire key simultaneously, rather than piece by piece.

The Vigenère and substitution ciphers do not have the properties of diffusion and confusion, which is why they are so susceptible to frequency analysis.

The concepts of diffusion and confusion play a role in any well-designed block cipher. Of course, a disadvantage (which is precisely the cryptographic advantage) of diffusion is error propagation: A small error in the ciphertext becomes a major error in the decrypted message, and usually means the decryption is unreadable.

The natural way of using a block cipher is to convert blocks of plaintext to blocks of ciphertext, independently and one at a time. This is called the electronic codebook (ECB) mode. Although it seems like the obvious way to implement a block cipher, we'll see that it is insecure and

that there are much better ways to use a block cipher. For example, it is possible to use feedback from the blocks of ciphertext in the encryption of subsequent blocks of plaintext. This leads to the cipher block chaining (CBC) mode and cipher feedback (CFB) mode of operation. These are discussed in [Section 6.3](#).

For an extensive discussion of block ciphers, see [Schneier].

6.2 Hill Ciphers

This section is not needed for understanding the rest of the chapter. It is included as an example of a block cipher.

In this section, we discuss the Hill cipher, which is a block cipher invented in 1929 by Lester Hill. It seems never to have been used much in practice. Its significance is that it was perhaps the first time that algebraic methods (linear algebra, modular arithmetic) were used in cryptography in an essential way. As we'll see in later chapters, algebraic methods now occupy a central position in the subject.

Choose an integer n , for example $n = 3$. The key is an $n \times n$ matrix M whose entries are integers mod 26. For example, let

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 11 & 9 & 8 \end{pmatrix}.$$

The message is written as a series of row vectors. For example, if the message is abc , we change this to the single row vector $(0, 1, 2)$. To encrypt, multiply the vector by the matrix (traditionally, the matrix appears on the right in the multiplication; multiplying on the left would yield a similar theory) and reduce mod 26:

$$(0, 1, 2) \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 11 & 9 & 8 \end{pmatrix} \equiv (0, 23, 22) \pmod{26}.$$

Therefore, the ciphertext is AXW . (The fact that the first letter a remained unchanged is a random occurrence; it is not a defect of the method.)

In order to decrypt, we need the determinant of M to satisfy

$$\gcd(\det(M), 26) = 1.$$

This means that there is a matrix N with integer entries such that $MN \equiv I \pmod{26}$, where I is the $n \times n$ identity matrix.

In our example, $\det(M) = -3$. The inverse of M is

$$\frac{-1}{3} \begin{pmatrix} -14 & 11 & -3 \\ 34 & -25 & 6 \\ -19 & 13 & -3 \end{pmatrix}.$$

Since 17 is the inverse of $-3 \pmod{26}$, we replace $-1/3$ by 17 and reduce mod 26 to obtain

$$N = \begin{pmatrix} 22 & 5 & 1 \\ 6 & 17 & 24 \\ 15 & 13 & 1 \end{pmatrix}.$$

The reader can check that $MN \equiv I \pmod{26}$.

For more on finding inverses of matrices mod n , see [Section 3.8](#). See also [Example 15](#) in the Computer Appendices.

The decryption is accomplished by multiplying by N , as follows:

$$(0, 23, 22) \begin{pmatrix} 22 & 5 & 1 \\ 6 & 17 & 24 \\ 15 & 13 & 1 \end{pmatrix} \equiv (0, 1, 2) \pmod{26}$$

In the general method with an $n \times n$ matrix, break the plaintext into blocks of n characters and change each block to a vector of n integers between 0 and 25 using $a = 0, b = 1, \dots, z = 25$. For example, with the matrix M as above, suppose our plaintext is

blockcipher.

This becomes (we add an x to fill the last space)

$$1 \ 11 \ 14 \quad 2 \ 10 \ 2 \quad 8 \ 15 \ 7 \quad 4 \ 17 \ 23.$$

Now multiply each vector by M , reduce the answer mod 26, and change back to letters:

$$\begin{aligned}(1, 11, 14)M &= (199, 183, 181) \equiv (17, 1, 25) \pmod{26} = RBZ \\ (2, 10, 2)M &= (64, 72, 82) \equiv (12, 20, 4) \pmod{26} = MUE,\end{aligned}$$

etc.

In our case, the ciphertext is

$$RBZMUEPYONOM.$$

It is easy to see that changing one letter of plaintext will usually change n letters of ciphertext. For example, if *block* is changed to *clock*, the first three letters of ciphertext change from *RBZ* to *SDC*. This makes frequency counts less effective, though they are not impossible when n is small. The frequencies of two-letter combinations, called **digrams**, and three-letter combinations, **trigrams**, have been computed. Beyond that, the number of combinations becomes too large (though tabulating the results for certain common combinations would not be difficult). Also, the frequencies of combinations are so low that it is hard to get meaningful data without a very large amount of text.

Now that we have the ciphertext, how do we decrypt? Simply break the ciphertext into blocks of length n , change each to a vector, and multiply on the right by the inverse matrix N . In our example, we have

$$RBZ = (17, 1, 25) \mapsto (17, 1, 25)N = (755, 427, 66) \equiv (1, 11, 14) = blo,$$

and similarly for the remainder of the ciphertext.

For another example, see [Example 21](#) in the Computer Appendices.

The Hill cipher is difficult to decrypt using only the ciphertext, but it succumbs easily to a known plaintext attack. If we do not know n , we can try various values until we find the right one. So suppose n is known. If we

have n of the blocks of plaintext of size n , then we can use the plaintext and the corresponding ciphertext to obtain a matrix equation for M (or for N , which might be more useful). For example, suppose we know that $n = 2$ and we have the plaintext

$$7 \quad 14 \quad 22 \quad 0 \quad 17 \quad 4 \quad 24 \quad 14 \quad 20 \quad 19 \quad 14 \quad 3 \quad 0 \quad 24$$

corresponding to the ciphertext

$$25 \quad 22 \quad 18 \quad 4 \quad 13 \quad 8 \quad 20 \quad 18 \quad 15 \quad 11 \quad 9 \quad 21 \quad 4 \quad 20$$

The first two blocks yield the matrix equation

$$\begin{pmatrix} 7 & 14 \\ 22 & 0 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} 25 & 22 \\ 18 & 4 \end{pmatrix} \pmod{26}.$$

Unfortunately, the matrix $\begin{pmatrix} 7 & 14 \\ 22 & 0 \end{pmatrix}$ has determinant -308 , which is not invertible mod 26 (though this matrix could be used to reduce greatly the number of choices for the encryption matrix). Therefore, we replace the last row of the equation, for example, by the fifth block to obtain

$$\begin{pmatrix} 7 & 14 \\ 20 & 19 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} 25 & 22 \\ 15 & 11 \end{pmatrix} \pmod{26}.$$

In this case, the matrix $\begin{pmatrix} 7 & 14 \\ 20 & 19 \end{pmatrix}$ is invertible mod 26 :

$$\begin{pmatrix} 7 & 14 \\ 20 & 19 \end{pmatrix}^{-1} \equiv \begin{pmatrix} 5 & 10 \\ 18 & 21 \end{pmatrix} \pmod{26}.$$

We obtain

$$M \equiv \begin{pmatrix} 5 & 10 \\ 18 & 21 \end{pmatrix} \begin{pmatrix} 25 & 22 \\ 15 & 11 \end{pmatrix} \equiv \begin{pmatrix} 15 & 12 \\ 11 & 3 \end{pmatrix} \pmod{26}.$$

Because the Hill cipher is vulnerable to this attack, it cannot be regarded as being very strong.

A chosen plaintext attack proceeds by the same strategy, but is a little faster. Again, if you do not know n , try various possibilities until one works. So suppose n is known. Choose the first block of plaintext to be $baaa \dots = 1000 \dots$, the second to be $abaa \dots = 0100 \dots$, and continue through the n th block being $\dots aaab = \dots 0001$. The blocks of ciphertext will be the rows of the matrix M .

For a chosen ciphertext attack, use the same strategy as for chosen plaintext, where the choices now represent ciphertext. The resulting plaintext will be the rows of the inverse matrix N .

6.3 Modes of Operation

Suppose we have a block cipher. It can encrypt a block of plaintext of a fixed size, for example 64 bits. There are many circumstances, however, where it is necessary to encrypt messages that are either longer or shorter than the cipher's block length. For example, a bank may be sending a terabyte of data to another bank. Or you might be sending a short message that needs to be encrypted one letter at a time since you want to produce ciphertext output as quickly as you write the plaintext input.

Block ciphers can be run in many different modes of operation, allowing users to choose appropriate modes to meet the requirements of their applications. There are five common modes of operation: electronic codebook (ECB), cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter (CTR) modes. We now discuss these modes.

6.3.1 Electronic Codebook (ECB)

The natural manner for using a block cipher is to break a long piece of plaintext into appropriately sized blocks of plaintext and process each block separately with the encryption function E_K . This is known as the electronic codebook (ECB) mode of operation. The plaintext P is broken into smaller chunks $P = [P_1, P_2, \dots, P_L]$ and the ciphertext is

$$C = [C_1, C_2, \dots, C_L]$$

where $C_j = E_K(P_j)$ is the encryption of P_j using the key K .

There is a natural weakness in the ECB mode of operation that becomes apparent when dealing with long pieces of plaintext. Say an adversary Eve has been observing communication between Alice and Bob for a long enough period of time. If Eve has managed to acquire some plaintext pieces corresponding to the ciphertext pieces that she has observed, she can start to build up a codebook with which she can decipher future communication between Alice and Bob. Eve never needs to calculate the key K ; she just looks up a ciphertext message in her codebook and uses the corresponding plaintext (if available) to decipher the message.

This can be a serious problem since many real-world messages consist of repeated fragments. E-mail is a prime example. An e-mail between Alice and Bob might start with the following header:

Date: Tue, 29 Feb 2000 13:44:38 -0500 (EST)

The ciphertext starts with the encrypted version of “Date: Tu”. If Eve finds that this piece of ciphertext often occurs on a Tuesday, she might be able to guess, without

knowing any of the plaintext, that such messages are e-mail sent on Tuesdays. With patience and ingenuity, Eve might be able to piece together enough of the message's header and trailer to figure out the context of the message. With even greater patience and computer memory, she might be able to piece together important pieces of the message.

Another problem that arises in ECB mode occurs when Eve tries to modify the encrypted message being sent to Bob. She might be able to extract important portions of the message and use her codebook to construct a false ciphertext message that she can insert in the data stream.

6.3.2 Cipher Block Chaining (CBC)

One method for reducing the problems that occur in ECB mode is to use chaining. Chaining is a feedback mechanism where the encryption of a block depends on the encryption of previous blocks.

In particular, encryption proceeds as

$$C_j = E_K(P_j \oplus C_{j-1}),$$

while decryption proceeds as

$$P_j = D_K(C_j) \oplus C_{j-1},$$

where C_0 is some chosen initial value. As usual, E_K and D_K denote the encryption and decryption functions for the block cipher.

Thus, in CBC mode, the plaintext is XORed with the previous ciphertext block and the result is encrypted.

Figure 6.1 depicts CBC.

Figure 6.1 Cipher Block Chaining Mode.

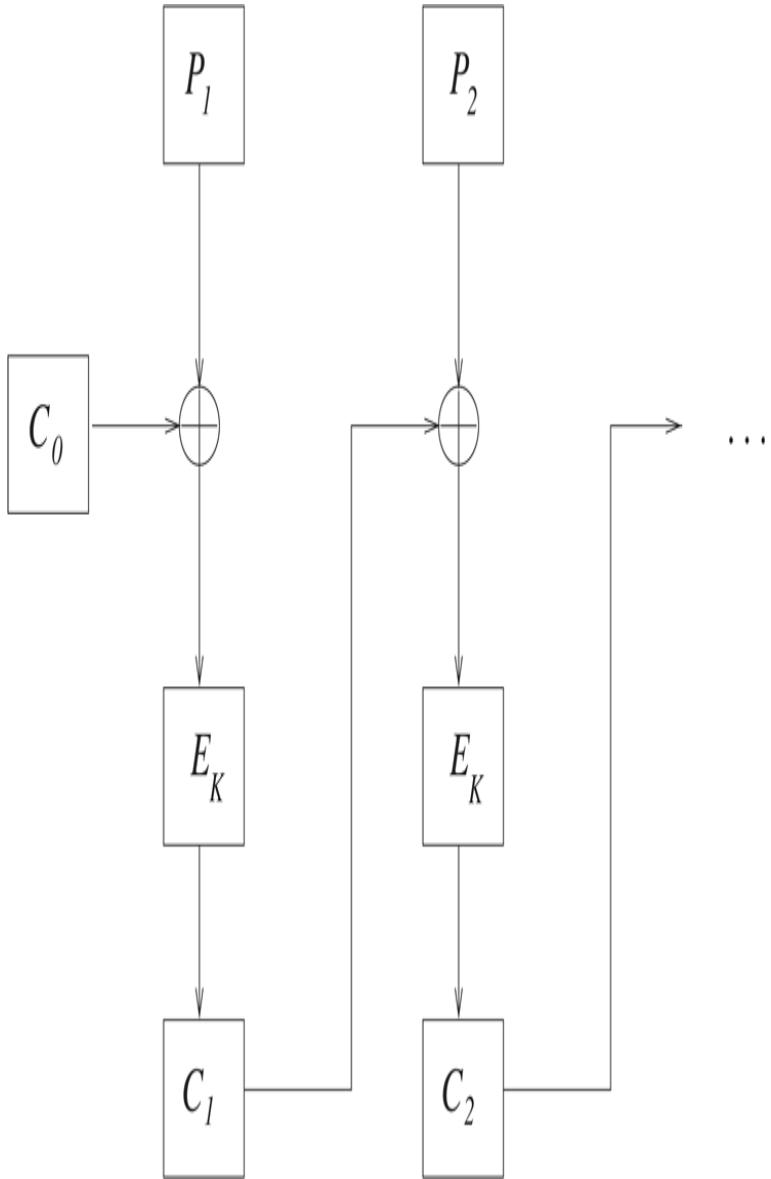


Figure 6.1 Full Alternative Text

The initial value C_0 is often called the initialization vector, or the IV. If we use a fixed value for C_0 , say $C_0 = 0$, and ever have the same plaintext message, the result will be that the resulting ciphertexts will be the same. This is undesirable since it allows the adversary to deduce that the same plaintext was created. This can be

very valuable information, and can often be used by the adversary to infer the meaning of the original plaintext.

In practice, this problem is handled by always choosing C_0 randomly and sending C_0 in the clear along with the first ciphertext C_1 . By doing so, even if the same plaintext message is sent repeatedly, an observer will see a different ciphertext each time.

6.3.3 Cipher Feedback (CFB)

One of the problems with both the CBC and ECB methods is that encryption (and hence decryption) cannot begin until a complete block of plaintext data is available. The cipher feedback mode operates in a manner that is very similar to the way in which LFSRs are used to encrypt plaintext. Rather than use linear recurrences to generate random bits, the cipher feedback mode is a stream mode of operation that produces pseudorandom bits using the block cipher E_K . In general, CFB operates in a k -bit mode, where each application produces k random bits for XORing with the plaintext. For our discussion, however, we focus on the eight-bit version of CFB. Using the eight-bit CFB allows one 8-bit piece of message (e.g., a single character) to be encrypted without having to wait for an entire block of data to be available. This is useful in interactive computer communications, for example.

For concreteness, let's assume that our block cipher encrypts blocks of 64 bits and outputs blocks of 64 bits (the sizes of the registers can easily be adjusted for other block sizes). The plaintext is broken into 8-bit pieces: $P = [P_1, P_2, \dots]$, where each P_j has eight bits, rather than the 64 bits used in ECB and CBC. Encryption proceeds as follows. An initial 64-bit X_1 is chosen. Then for $j = 1, 2, 3, \dots$, the following is performed:

$$\begin{aligned}O_j &= L_8(E_K(X_j)) \\C_j &= P_j \oplus O_j \\X_{j+1} &= R_{56}(X_j) \parallel C_j,\end{aligned}$$

where $L_8(X)$ denotes the 8 leftmost bits of X , $R_{56}(X)$ denotes the rightmost 56 bits of X , and $X \parallel Y$ denotes the string obtained by writing X followed by Y . We present the CFB mode of operation in [Figure 6.2](#).

Figure 6.2 Cipher Feedback Mode.

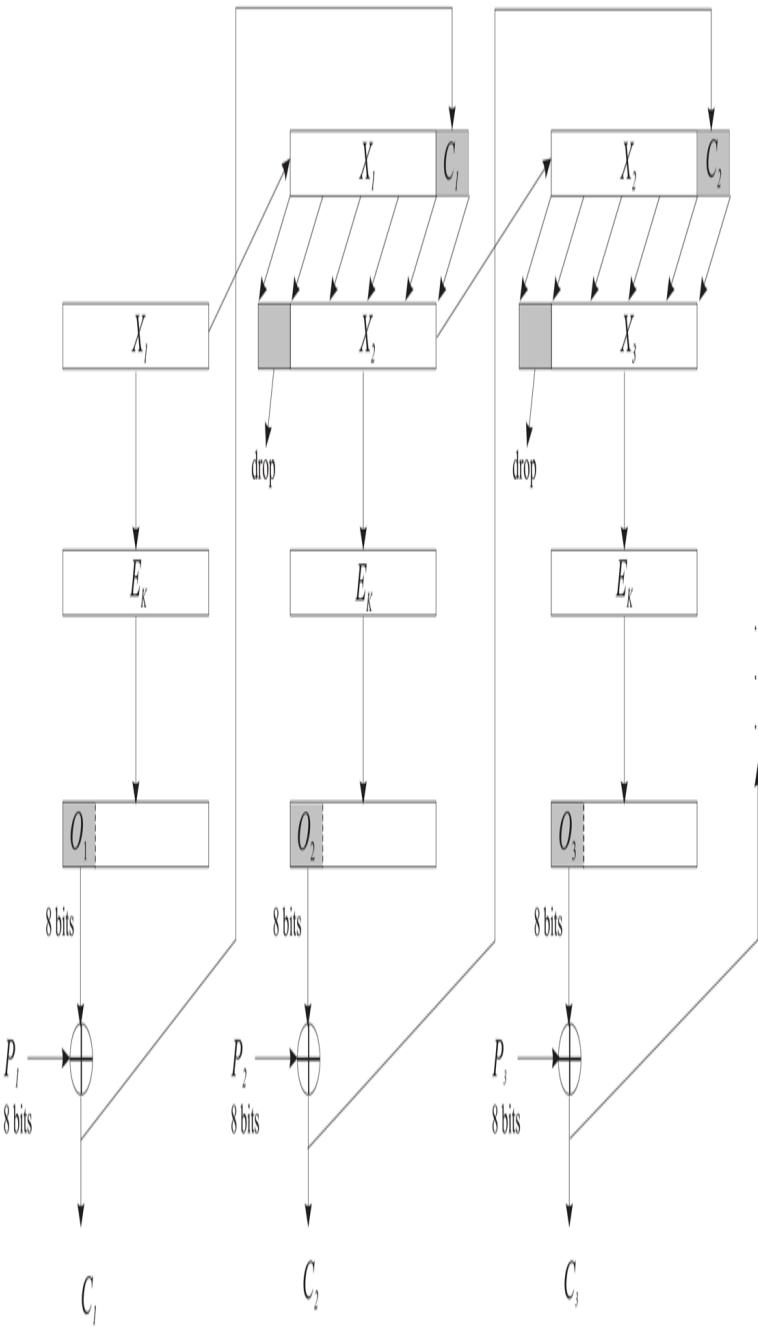


Figure 6.2 Full Alternative Text

Decryption is done with the following steps:

$$P_j = C_j \oplus L_8(E_K(X_j))$$

$$X_{j+1} = R_{56}(X_j) \parallel C_j.$$

We note that decryption does not involve the decryption function, D_K . This would be an advantage of running a block cipher in a stream mode in a case where the

decryption function for the block cipher is slower than the encryption function.

Let's step through one round of the CFB algorithm. First, we have a 64-bit register that is initialized with X_1 .

These 64 bits are encrypted using E_K and the leftmost eight bits of $E_K(X_1)$ are extracted and XORed with the 8-bit P_1 to form C_1 . Then C_1 is sent to the recipient.

Before working with P_2 , the 64-bit register X_1 is updated by extracting the rightmost 56 bits. The eight bits of C_1 are appended on the right to form

$X_2 = R_{56}(X_1) \parallel C_1$. Then P_2 is encrypted by the same process, but using X_2 in place of X_1 . After P_2 is encrypted to C_2 , the 64-bit register is updated to form

$$X_3 = R_{56}(X_2) \parallel C_2 = R_{48}(X_1) \parallel C_1 \parallel C_2.$$

By the end of the 8th round, the initial X_1 has disappeared from the 64-bit register and

$X_9 = C_1 \parallel C_2 \parallel \dots \parallel C_8$. The C_j continue to pass through the register, so for example

$$X_{20} = C_{12} \parallel C_{13} \parallel \dots \parallel C_{19}.$$

Note that CFB encrypts the plaintext in a manner similar to one-time pads or LFSRs. The key K and the numbers X_j are used to produce binary strings that are XORed with the plaintext to produce the ciphertext. This is a much different type of encryption than the ECB and CBC, where the ciphertext is the output of DES.

In practical applications, CFB is useful because it can recover from errors in transmission of the ciphertext.

Suppose that the transmitter sends the ciphertext blocks $C_1, C_2, \dots, C_k, \dots$, and C_1 is corrupted during transmission, so that the receiver observes \tilde{C}_1, C_2, \dots

Decryption takes \tilde{C}_1 and produces a garbled version of P_1 with bit errors in the locations that \tilde{C}_1 had bit errors.

Now, after decrypting this ciphertext block, the receiver forms an incorrect X_2 , which we denote \tilde{X}_2 . If X_1 was $(*, *, *, *, *, *, *, *, *)$, then

$\tilde{X}_2 = (*, *, *, *, *, *, *, \tilde{C}_1)$. When the receiver gets an uncorrupted C_2 and decrypts, then a completely garbled version of P_2 is produced. When forming X_3 , the decrypter actually forms

$\tilde{X}_3 = (*, *, *, *, *, *, \tilde{C}_1, C_2)$. The decrypter repeats this process, ultimately getting bad versions of P_1, P_2, \dots, P_9 . When the decrypter calculates X_9 , the error block has moved to the leftmost block of \tilde{X}_9 as

$\tilde{X}_9 = (\tilde{C}_1, C_2, \dots, C_8)$. At the next step, the error will have been flushed from the X_{10} register, and X_{10} and subsequent registers will be uncorrupted. For a simplified version of these ideas, see [Exercise 18](#).

6.3.4 Output Feedback (OFB)

The CBC and CFB modes of operation exhibit a drawback in that errors propagate for a duration of time corresponding to the block size of the cipher. The output feedback mode (OFB) is another example of a stream mode of operation for a block cipher where encryption is performed by XORing the message with a pseudorandom bit stream generated by the block cipher. One advantage of the OFB mode is that it avoids error propagation.

Much like CFB, OFB may work on chunks of different sizes. For our discussion, we focus on the eight-bit version of OFB, where OFB is used to encrypt eight-bit chunks of plaintext in a streaming mode. Just as for CFB, we break our plaintext into eight-bit pieces, with $P = [P_1, P_2, \dots]$. We start with an initial value X_1 , which has a length equal to the block length of the cipher, for example, 64 bits (the sizes of the registers can easily be adjusted for other block sizes). X_1 is often called the IV, and should be chosen to be random. X_1 is encrypted using the key K to produce 64 bits of output, and the leftmost eight bits O_1 of the ciphertext are extracted. These are then XORed with the first eight bits

P_1 of the plaintext to produce eight bits of ciphertext, C_1

So far, this is the same as what we were doing in CFB.
But OFB differs from CFB in what happens next. In order to iterate, CFB updates the register X_2 by extracting the right 56 bits of X_1 and appending C_1 to the right side.
Rather than use the ciphertext, OFB uses the output of the encryption. That is, OFB updates the register X_2 by extracting the right 56 bits of X_1 and appending O_1 to the right side.

In general, the following procedure is performed for $j = 1, 2, 3, \dots$:

$$\begin{aligned} O_j &= L_8(E_K(X_j)) \\ X_{j+1} &= R_{56}(X_j) \parallel O_j \\ C_j &= P_j \oplus O_j. \end{aligned}$$

We depict the steps for the OFB mode of operation in [Figure 6.3](#). Here, the output stream O_j is the encryption of the register containing the previous output from the block cipher. This output is then treated as a keystream and is XORed with the incoming plaintexts P_j to produce a stream of ciphertexts. Decryption is very simple. We get the plaintext P_j by XORing the corresponding ciphertext C_j with the output keystream O_j . Again, just like CFB, we do not need the decryption function D_K .

Figure 6.3 Output Feedback Mode.

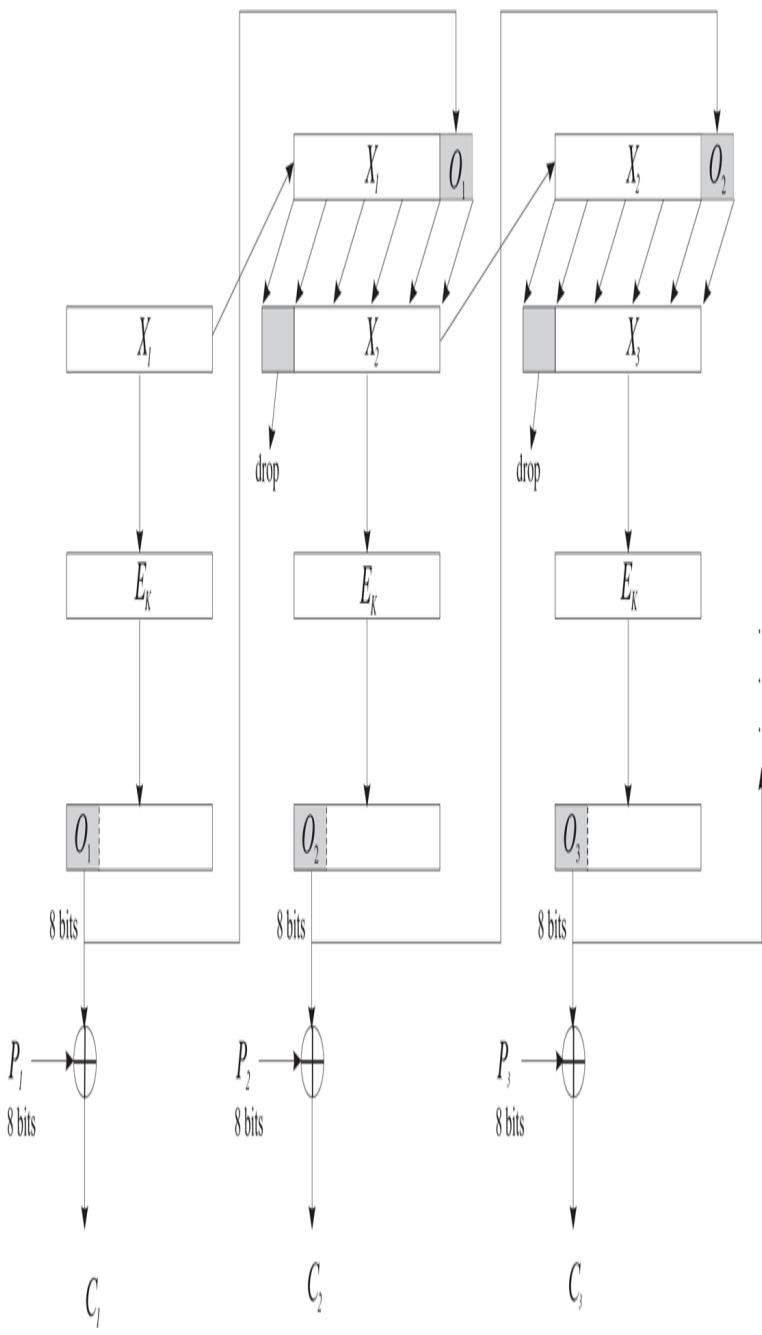


Figure 6.3 Full Alternative Text

So why would one want to build a stream cipher this way as opposed to the way the CFB stream cipher was built? There are a few key advantages to the OFB strategy. First, the generation of the O_j output key stream may be performed completely without any plaintext. What this means is that the key stream can be generated in advance. This might be desirable for applications where

we cannot afford to perform encryption operations as the plaintext message arrives.

Another advantage lies in its performance when errors are introduced to the ciphertext. Suppose a few errors are introduced to C_j when it is delivered to the receiver. Then only those corresponding bits in the plaintext are corrupted when decryption is performed. Since we build future output streams using the encryption of the register, and not using the corrupted ciphertext, the output stream will always remain clean and the errors in the ciphertext will not propagate.

To summarize, CFB required the register to completely flush itself of errors, which produced an entire block length of garbled plaintext bits. OFB, on the other hand, will immediately correct itself.

There is one problem associated with OFB, however, that is common to all stream ciphers that are obtained by XORing pseudorandom numbers with plaintext. If Eve knows a particular plaintext P_j and ciphertext C_j , she can conduct the following attack. She first calculates

$$O_j = C_j \oplus P_j$$

to get out the key stream. She may then create any false plaintext P'_j she wants. Now, to produce a ciphertext, she merely has to XOR with the output stream she calculated:

$$C'_j = P'_j \oplus O_j.$$

This allows her to modify messages.

6.3.5 Counter (CTR)

The counter (CTR) mode builds upon the ideas that were used in the OFB mode. Just like OFB, CTR creates an output key stream that is XORed with chunks of

plaintext to produce ciphertext. The main difference between CTR and OFB lies in the fact that the output stream O_j in CTR is not linked to previous output streams.

CTR starts with the plaintext broken into eight-bit pieces, $P = [P_1, P_2, \dots]$. We begin with an initial value X_1 , which has a length equal to the block length of the cipher, for example, 64 bits. Now, X_1 is encrypted using the key K to produce 64 bits of output, and the leftmost eight bits of the ciphertext are extracted and XORed with P_1 to produce eight bits of ciphertext, C_1 .

Now, rather than update the register X_2 to contain the output of the block cipher, we simply take

$X_2 = X_1 + 1$. In this way, X_2 does not depend on previous output. CTR then creates new output stream by encrypting X_2 . Similarly, we may proceed by using $X_3 = X_2 + 1$, and so on. The j th ciphertext is produced by XORing the left eight bits from the encryption of the j th register with the corresponding plaintext P_j .

In general, the procedure for CTR is

$$\begin{aligned} X_j &= X_{j-1} + 1 \\ O_j &= L_8(E_K(X_j)) \\ C_j &= P_j \oplus O_j \end{aligned}$$

for $j = 2, 3, \dots$, and is presented in Figure 6.4. The reader might wonder what happens to X_j if we continually add 1 to it. Shouldn't it eventually become too large? This is unlikely to happen, but if it does, we simply wrap around and start back at 0.

Figure 6.4 Counter Mode.

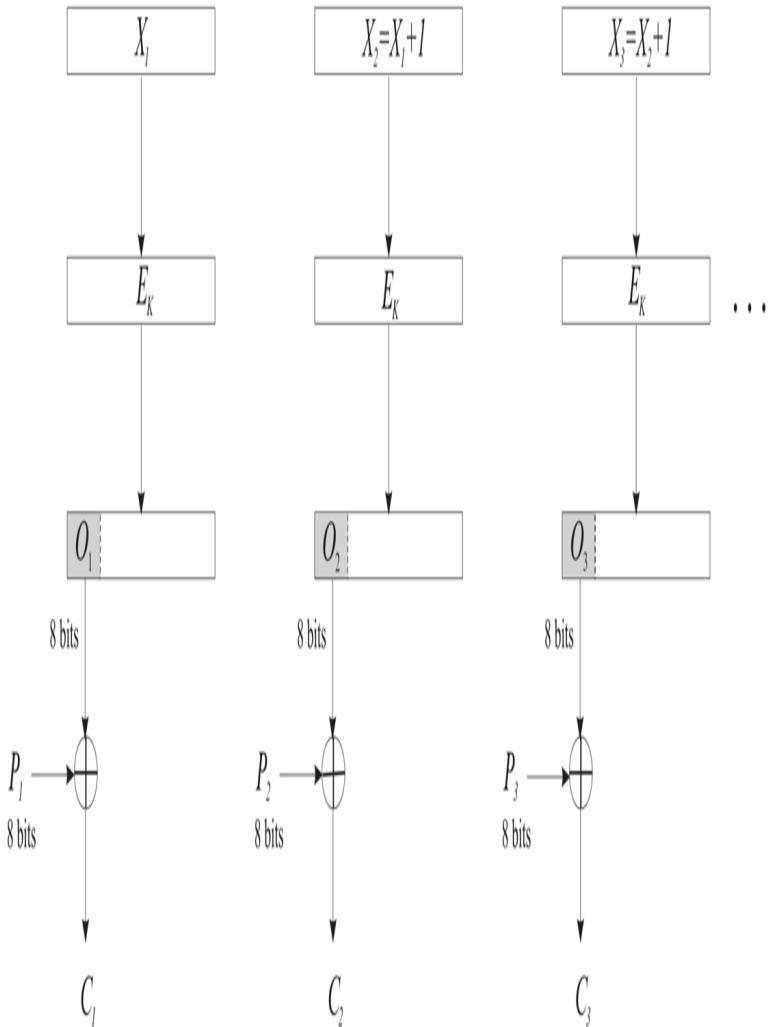


Figure 6.4 Full Alternative Text

Just like OFB, the registers X_j can be calculated ahead of time, and the actual encryption of plaintext is simple in that it involves just the XOR operation. As a result, its performance is identical to OFB's when errors are introduced in the ciphertext. The advantage of CTR mode compared to OFB, however, stems from the fact that many output chunks O_j may be calculated in parallel. We do not have to calculate O_j before calculating O_{j+1} . This makes CTR mode ideal for parallelizing.

6.4 Multiple Encryption

As technology improves and more sophisticated attacks are developed, encryption systems become less secure and need to be replaced. There are two main approaches to achieving increased security. The first involves using encryption multiple times and leads, for example, to triple encryption. The second approach is to find a new system that is more secure, a potentially lengthy process.

We start by describing the idea behind multiple encryption schemes. The idea is to encrypt the same plaintext multiple times using the same algorithm with different keys. **Double encryption** encrypts the plaintext by first encrypting with one key and then encrypting again using another key. For example, if the keyspace for single encryption has 56 bits, hence 2^{56} keys, then the new keyspace consists of 2^{112} keys. One might guess that double encryption should therefore double the security. This, however, is not true. Merkle and Hellman showed that the double encryption scheme actually has the security level of a 57-bit key. The reduction from 2^{112} to 2^{57} makes use of the **meet-in-the-middle attack**, which is described in the next section.

Since double encryption has a weakness, **triple encryption** is often used. This appears to have a level of security approximately equivalent to a 112-bit key (when the single encryption has a 56-bit key). There are at least two ways that triple encryption can be implemented. One is to choose three keys, K_1 , K_2 , K_3 , and perform $E_{K_1}(E_{K_2}(E_{K_3}(m)))$. This type of triple encryption is sometimes called *EEE*. The other is to choose two keys, K_1 and K_2 , and perform $E_{K_1}(D_{K_2}(E_{K_1}(m)))$. This is sometimes called *EDE*. When $K_1 = K_2$, this reduces to

single encryption. Therefore, a triple encryption machine that is communicating with an older machine that still uses single encryption can simply set $K_1 = K_2$ and proceed. This compatibility is the reason for using D_{K_2} instead of E_{K_2} in the middle; the use of D instead of E gives no extra cryptographic strength. Both versions of triple encryption are resistant to meet-in-the-middle attacks (compare with [Exercise 11](#)). However, there are other attacks on the two-key version ([Merkle-Hellman] and [van Oorschot-Wiener]) that indicate possible weaknesses, though they require so much memory as to be impractical.

Another strengthening of encryption was proposed by Rivest. Choose three keys, K_1 , K_2 , K_3 , and perform $K_3 \oplus E_{K_2}(K_1 \oplus m)$. In other words, modify the plaintext by *XOR*ing with K_1 , then apply encryption with K_2 , then *XOR* the result with K_3 . This method, when used with DES, is known as DESX and has been shown to be fairly secure. See [Kilian-Rogaway].

6.5 Meet-in-the-Middle Attacks

Alice and Bob are using an encryption method. The encryption functions are called E_K , and the decryption functions are called D_K , where K is a key. We assume that if someone knows K , then she also knows E_K and D_K (so Alice and Bob could be using one of the classical, nonpublic key systems such as DES or AES). They have a great idea. Instead of encrypting once, they use two keys K_1 and K_2 and encrypt twice. Starting with a plaintext message m , the ciphertext is $c = E_{K_2}(E_{K_1}(m))$. To decrypt, simply compute $m = D_{K_1}(D_{K_2}(c))$. Eve will need to discover both K_1 and K_2 to decrypt their messages.

Does this provide greater security? For many cryptosystems, applying two encryptions is the same as using an encryption for some other key. For example, the composition of two affine functions is still an affine function (see [Exercise 11 in Chapter 2](#)). Similarly, using two RSA encryptions (with the same n) with exponents e_1 and e_2 corresponds to doing a single encryption with exponent e_1e_2 . In these cases, double encryption offers no advantage. However, there are systems, such as DES (see [Subsection 7.4.1](#)) where the composition of two encryptions is not simply encryption with another key. For these, double encryption might seem to offer a much higher level of security. However, the following attack shows that this is not really the case, as long as we have a computer with a lot of memory.

Assume Eve has intercepted a message m and a doubly encrypted ciphertext $c = E_{K_2}(E_{K_1}(m))$. She wants to find K_1 and K_2 . She first computes two lists:

1. $E_K(m)$ for all possible keys K
2. $D_L(c)$ for all possible keys L .

Finally, she compares the two lists and looks for matches. There will be at least one match, since the correct pair of keys will be one of them, but it is likely that there will be many matches. If there are several matches, she then takes another plaintext–ciphertext pair and determines which of the pairs (K, L) she has found will encrypt the plaintext to the ciphertext. This should greatly reduce the list. If there is still more than one pair remaining, she continues until only one pair remains (or she decides that two or more pairs give the same double encryption function). Eve now has the desired pair K_1, K_2 .

If Eve has only one plaintext–ciphertext pair, she still has reduced the set of possible key pairs to a short list. If she intercepts a future transmission, she can try each of these possibilities and obtain a very short list of meaningful plaintexts.

If there are N possible keys, Eve needs to compute N values $E_L(m)$. She then needs to compute N numbers $D_L(c)$ and compare them with the stored list. But these $2N$ computations (plus the comparisons) are much less than the N^2 computations required for searching through all key pairs K_1, K_2 .

This meet-in-the-middle procedure takes slightly longer than the exhaustive search through all keys for single encryption. It also takes a lot of memory to store the first list. However, the conclusion is that double encryption does not significantly raise the level of security.

Similarly, we could use triple encryption, using triples of keys. A similar attack brings the level of security down to at most what one might naively expect from double

encryption, namely squaring the possible number of keys.

Example

Suppose the single encryption has 2^{56} possible keys and the block cipher inputs and outputs blocks of 64 bits, as is the case with DES. The first list has 2^{56} entries, each of which is a 64-bit block. The probability that a given block in List 1 matches a given block in List 2 is 2^{-64} . Since there are 2^{56} entries in List 2, we expect that a given block in List 1 matches $2^{56}2^{-64} = 2^{-8}$ entries of List 2. Running through the 2^{56} elements in List 1, we expect $2^{56}2^{-8} = 2^{48}$ pairs (K, L) for which there are matches between List 1 and List 2.

We know that one of these 2^{48} matches is from the correct pair (K_1, K_2) , and the other matches are probably caused by randomness. If we take a random pair (K, L) and try it on a new plaintext–ciphertext (m_1, c_1) , then $E_L(E_K(m_1))$ is a 64-bit block that has probability 2^{-64} of matching the 64-bit block c_1 . Therefore, among the approximately 2^{48} random pairs, we expect $2^{48}2^{-64} = 2^{-16}$ matches between $E_L(E_K(m_1))$ and c_1 . In other words, it is likely that the second plaintext–ciphertext pair eliminates all extraneous solutions and leaves only the correct key pair (K_1, K_2) . If not, a third round should complete the task.

6.6 Exercises

1. The ciphertext $YIFZMA$ was encrypted by a Hill cipher with matrix $\begin{pmatrix} 9 & 13 \\ 2 & 3 \end{pmatrix}$. Find the plaintext.
2. The matrix $\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \pmod{26}$ is not suitable for the matrix in a Hill cipher. Why?
3. The ciphertext text $GEZXDS$ was encrypted by a Hill cipher with a 2×2 matrix. The plaintext is *solved*. Find the encryption matrix M .
4. Consider the following combination of Hill and Vigenère ciphers: The key consists of three 2×2 matrices, M_1, M_2, M_3 . The plaintext letters are represented as integers mod 26. The first two are encrypted by M_1 , the next two by M_2 , the 5th and 6th by M_3 . This is repeated cyclically, as in the Vigenère cipher. Explain how to do a chosen plaintext attack on this system. Assume that you know that three 2×2 matrices are being used. State explicitly what plaintexts you would use and how you would use the outputs.
5. Eve captures Bob's Hill cipher machine, which uses a 2-by-2 matrix $M \pmod{26}$. She tries a chosen plaintext attack. She finds that the plaintext ba encrypts to HC and the plaintext zz encrypts to GT . What is the matrix M ?
6. Alice uses a Hill cipher with a 3×3 matrix M that is invertible mod 26. Describe a chosen plaintext attack that will yield the entries of the matrix M . Explicitly say what plaintexts you will use.
7.
 1. The ciphertext text $ELNI$ was encrypted by a Hill cipher with a 2×2 matrix. The plaintext is *dont*. Find the encryption matrix.
 2. Suppose the ciphertext is $ELNK$ and the plaintext is still *dont*. Find the encryption matrix. Note that the second column of the matrix is changed. This shows that the entire second column of the encryption matrix is involved in obtaining the last character of the ciphertext.
8. Suppose the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ is used for an encryption matrix in a Hill cipher. Find two plaintexts that encrypt to the same ciphertext.

9. Let a, b, c, d, e, f be integers mod 26. Consider the following combination of the Hill and affine ciphers: Represent a block of plaintext as a pair (x, y) mod 26. The corresponding ciphertext (u, v) is

$$(x \quad y) \begin{pmatrix} a & b \\ c & d \end{pmatrix} + (e \quad f) \equiv (u \quad v) \pmod{26}.$$

Describe how to carry out a chosen plaintext attack on this system (with the goal of finding the key a, b, c, d, e, f). You should state explicitly what plaintexts you choose and how to recover the key.

10. Alice is sending a message to Bob using a Hill cipher with a 2×2 matrix. In fact, Alice is bored and her plaintext consists of the letter a repeated a few hundred times. Eve knows what system is being used, but not the key, and intercepts the ciphertext. State how Eve will recognize that the plaintext is one repeated letter and decide whether or not Eve can deduce the letter and/or the key.
(Note: The solution very much depends on the fact that the repeated letter is a , rather than b, c, \dots)
11. Let E_K denote encryption (for some cryptosystem) with key K . Suppose that there are 2^{35} possible keys K . Alice decides to encrypt a message m as follows:

She chooses two keys K and L and double encrypts by computing

$$c = E_K(E_K(E_L(m)))$$

to get the ciphertext c . Suppose Eve knows Alice's method of encryption (but not K and L) and has at least two plaintext–ciphertext pairs. Describe a method that is *guaranteed* to yield the correct K and L (and maybe a *very small* additional set of incorrect pairs). Be explicit enough to say why you are using at least two plaintext–ciphertext pairs. Eve may do up to 2^{50} computations.

12. Alice and Bob are arguing about which method of multiple encryption they should use. Alice wants to choose keys K_1 and K_2 and triple encrypt a message m as
 $c = E_{K_1}(E_{K_2}(E_{K_1}(m)))$. Bob wants to encrypt m as
 $c = E_{K_1}(E_{K_1}(E_{K_2}(E_{K_2}(m))))$. Which method is more secure? Describe in detail an attack on the weaker encryption method.
13. Alice and Bob are trying to implement triple encryption. Let E_K denote DES encryption with key K and let D_K denote decryption.
1. Alice chooses two keys, K_1 and K_2 , and encrypts using the formula $c = E_{K_1}(D_{K_2}(E_{K_1}(m)))$. Bob chooses two keys, K_1 and K_2 , and encrypts using the formula $c = E_{K_1}(E_{K_2}(E_{K_2}(m)))$. One of these methods is more secure than the other. Say which one is weaker and explicitly give the steps that can be used to attack the

weaker system. You may assume that you know ten plaintext–ciphertext pairs.

2. What is the advantage of using D_{K_2} instead of E_{K_2} in Alice's version?

14. Suppose E^1 and E^2 are two encryption methods. Let K_1 and K_2 be keys and consider the double encryption

$$E_{K_1, K_2}(m) = E_{K_1}^1(E_{K_2}^2(m)).$$

1. Suppose you know a plaintext–ciphertext pair. Show how to perform a meet-in-the-middle attack on this double encryption.

2. An affine encryption given by $x \mapsto \alpha x + \beta \pmod{26}$ can be regarded as a double encryption, where one encryption is multiplying the plaintext by α and the other is a shift by β . Assume that you have a plaintext and ciphertext that are long enough that α and β are unique. Show that the meet-in-the-middle attack from part (a) takes at most 38 steps (not including the comparisons between the lists). Note that this is much faster than a brute force search through all 312 keys.

15. Let E_K denote DES encryption with key K . Suppose there is a public database Y consisting of 10^{10} DES keys and there is another public database Z of 10^{10} binary strings of length 64. Alice has five messages m_1, m_2, \dots, m_5 . She chooses a key K from Y and a string B from Z . She encrypts each message m by computing $c = E_K(m) \oplus B$. She uses the same K and B for each of the messages. She shows the five plaintext–ciphertext pairs (m_i, c_i) to Eve and challenges Eve to find K and B . Alice knows that Eve's computer can do only 10^{15} calculations, and there are 10^{20} pairs (K, B) , so Alice thinks that Eve cannot find the correct pair. However, Eve has taken a crypto course. Show how she can find the K and B that Alice used. You must state explicitly what Eve does. Statements such as "Eve makes a list" are not sufficient; you must include what is on the lists and how long they are.

16. Alice wants to encrypt her messages securely, but she can afford only an encryption machine that uses a 25-bit key. To increase security, she chooses 4 keys K_1, K_2, K_3, K_4 and encrypts four times:

$$c = E_{K_1}(E_{K_2}(E_{K_3}(E_{K_4}(m))).$$

Eve finds several plaintext–ciphertext pairs (m, c) encrypted with this set of keys. Describe how she can find (with high probability) the keys K_1, K_2, K_3, K_4 . (For this problem, assume that Eve can do at most 2^{60} computations, so she cannot try all 2^{100} combinations of keys.) (Note: If you use only one of the plaintext–

ciphertext pairs in your solution, you probably have not done enough to determine the keys.)

17. Show that the decryption procedures given for the CBC and CFB modes actually perform the desired decryptions.

18. Consider the following simplified version of the CFB mode. The plaintext is broken into 32-bit pieces: $P = [P_1, P_2, \dots]$, where each P_j has 32 bits, rather than the eight bits used in CFB. Encryption proceeds as follows. An initial 64-bit X_1 is chosen. Then for $j = 1, 2, 3, \dots$, the following is performed:

$$C_j = P_j \oplus L_{32}(E_K(X_j)) \\ X_{j+1} = R_{32}(X_j) \parallel C_j,$$

where $L_{32}(X)$ denotes the 32 leftmost bits of X , $R_{32}(X)$ denotes the rightmost 32 bits of X , and $X \parallel Y$ denotes the string obtained by writing X followed by Y .

1. Find the decryption algorithm.
2. The ciphertext consists of 32-bit blocks $C_1, C_2, C_3, C_4, \dots$. Suppose that a transmission error causes C_1 to be received as $\tilde{C}_1 \neq C_1$, but that C_2, C_3, C_4, \dots are received correctly. This corrupted ciphertext is then decrypted to yield plaintext blocks $\tilde{P}_1, \tilde{P}_2, \dots$. Show that $\tilde{P}_1 \neq P_1$, but that $\tilde{P}_i = P_i$ for all $i \geq 4$. Therefore, the error affects only three blocks of the decryption.
19. The cipher block chaining (CBC) mode has the property that it recovers from errors in ciphertext blocks. Show that if an error occurs in the transmission of a block C_j , but all the other blocks are transmitted correctly, then this affects only two blocks of the decryption. Which two blocks?
20. In CTR mode, the initial X_1 has 64 bits and is sent unencrypted to the receiver. (a) If X_1 is chosen randomly every time a message is encrypted, approximately how many messages must be sent in order for there to be a good chance that two messages use the same X_1 ? (b) What could go wrong if the same X_1 is used for two different messages? (Assume that the key K is not changed.)
21. Suppose that in CBC mode, the final plaintext block P_n is incomplete; that is, its length M is less than the usual block size of, say, 64 bits. Often, this last block is padded with a binary string to make it have full length. Another method that can be used is called **ciphertext stealing**, as follows:
 1. Compute $Y_{n-1} = E_K(C_{n-2} \oplus P_{n-1})$.
 2. Compute $C_n = L_M(Y_{n-1})$, where L_M means we take the leftmost M bits.

3. Compute $C_{n-1} = E_K((P_n || 0^{64-M}) \oplus Y_{n-1})$, where $P_n || 0^{64-M}$ denotes P_n with enough os appended to give it the length of a full 64-bit block.
4. The ciphertext is $C_1 C_2 \cdots C_{n-1} C_n$. Therefore, the ciphertext has the same length as the plaintext.

Suppose you receive a message that used this ciphertext stealing for the final blocks (the ciphertext blocks C_1, \dots, C_{n-2} were computed in the usual way for CBC). Show how to decrypt the ciphertext (you have the same key as the sender).

22. Suppose Alice has a block cipher with 2^{50} keys, Bob has one with 2^{40} keys, and Carla has one with 2^{30} keys. The only known way to break single encryption with each system is by brute force, namely trying all keys. Alice uses her system with single encryption. But Bob uses his with double encryption, and Carla uses hers with triple encryption. Who has the most secure system? Who has the weakest? (Assume that double and triple encryption do not reduce to using single or double encryption, respectively. Also, assume that some plaintext-ciphertext pairs are available for Alice's single encryption, Bob's double encryption, and Carla's triple encryption.)

6.7 Computer Problems

1. The following is the ciphertext of a Hill cipher

zirkzwopjjoptfapuhfhadrq

using the matrix

Decrypt.

Chapter 7 The Data Encryption Standard

7.1 Introduction

In 1973, the National Bureau of Standards (NBS), later to become the National Institute of Standards and Technology (NIST), issued a public request seeking a cryptographic algorithm to become a national standard. IBM submitted an algorithm called LUCIFER in 1974. The NBS forwarded it to the National Security Agency, which reviewed it and, after some modifications, returned a version that was essentially the Data Encryption Standard (DES) algorithm. In 1975, NBS released DES, as well as a free license for its use, and in 1977 NBS made it the official data encryption standard.

DES was used extensively in electronic commerce, for example in the banking industry. If two banks wanted to exchange data, they first used a public key method such as RSA to transmit a key for DES, then they used DES for transmitting the data. It had the advantage of being very fast and reasonably secure.

From 1975 on, there was controversy surrounding DES. Some regarded the key size as too small. Many were worried about NSA's involvement. For example, had they arranged for it to have a "trapdoor" – in other words, a secret weakness that would allow only them to break the system? It was also suggested that NSA modified the design to avoid the possibility that IBM had inserted a trapdoor in LUCIFER. In any case, the design decisions remained a mystery for many years.

In 1990, Eli Biham and Adi Shamir showed how their method of differential cryptanalysis could be used to attack DES, and soon thereafter they showed how these methods could succeed faster than brute force. This indicated that perhaps the designers of DES had been aware of this type of attack. A few years later, IBM released some details of the design criteria, which showed that indeed they had constructed the system to be resistant to differential cryptanalysis. This cleared up at least some of the mystery surrounding the algorithm.

DES lasted for a long time, but became outdated. Brute force searches (see [Section 7.5](#)), though expensive, can now break the system. Therefore, NIST replaced it with the system AES (see [Chapter 8](#)) in the year 2000. However, it is worth studying DES since it represents a popular class of algorithms and it was one of the most frequently used cryptographic algorithms in history.

DES is a block cipher; namely, it breaks the plaintext into blocks of 64 bits, and encrypts each block separately. The actual mechanics of how this is done is often called a **Feistel system**, after Horst Feistel, who was part of the IBM team that developed LUCIFER. In the next section, we give a simple algorithm that has many of the characteristics of this type of system, but is small enough to use as an example. In [Section 7.3](#), we show how differential cryptanalysis can be used to attack this simple system. We give the DES algorithm in [Section 7.4](#). Finally, in [Section 7.5](#), we describe some methods used to break DES.

7.2 A Simplified DES-Type Algorithm

The DES algorithm is rather unwieldy to use for examples, so in the present section we present an algorithm that has many of the same features, but is much smaller. Like DES, the present algorithm is a block cipher. Since the blocks are encrypted separately, we assume throughout the present discussion that the full message consists of only one block.

The message has 12 bits and is written in the form L_0R_0 , where L_0 consists of the first six bits and R_0 consists of the last six bits. The key K has nine bits. The i th round of the algorithm transforms an input $L_{i-1}R_{i-1}$ to the output L_iR_i using an eight-bit key K_i derived from K .

The main part of the encryption process is a function $f(R_{i-1}, K_i)$ that takes a six-bit input R_{i-1} and an eight-bit input K_i and produces a six-bit output. This will be described later.

The output for the i th round is defined as follows:

$$L_i = R_{i-1} \text{ and } R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

where \oplus denotes XOR, namely bit-by-bit addition mod 2. This is depicted in Figure 7.1.

Figure 7.1 One Round of a Feistel System

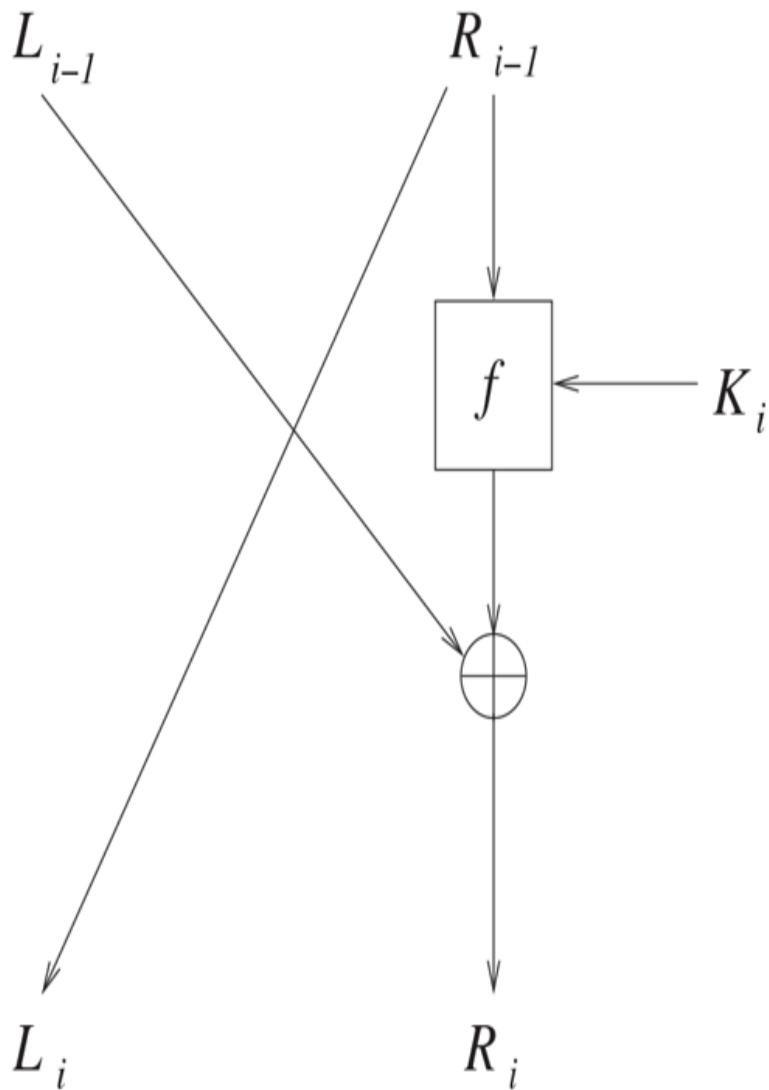


Figure 7.1 Full Alternative Text

This operation is performed for a certain number of rounds, say n , and produces the ciphertext L_nR_n .

How do we decrypt? Start with L_nR_n and switch left and right to obtain R_nL_n . (Note: *This switch is built into the DES encryption algorithm, so it is not needed when decrypting DES.*) Now use the same procedure as before, but with the keys K_i used in reverse order K_n, \dots, K_1 . Let's see how this works. The first step takes R_nL_n and gives the output

$$[L_n] \quad [R_n \oplus f(L_n, K_n)].$$

We know from the encryption procedure that

$$L_n = R_{n-1} \text{ and } R_n = L_{n-1} \oplus f(R_{n-1}, K_n).$$

Therefore,

$$\begin{aligned}[L_n] & [R_n \oplus f(L_n, K_n)] = [R_{n-1}] & [L_{n-1} \oplus f(R_{n-1}, K_n) \oplus f(L_n, K_n)] \\ & = [R_{n-1}] & [L_{n-1}].\end{aligned}$$

The last equality again uses $L_n = R_{n-1}$, so that

$f(R_{n-1}, K_n) \oplus f(L_n, K_n)$ is 0. Similarly, the second step of decryption sends $R_{n-1}L_{n-1}$ to $R_{n-2}L_{n-2}$.

Continuing, we see that the decryption process leads us back to R_0L_0 . Switching the left and right halves, we obtain the original plaintext L_0R_0 , as desired.

Note that the decryption process is essentially the same as the encryption process. We simply need to switch left and right and use the keys K_i in reverse order.

Therefore, both the sender and receiver use a common key and they can use identical machines (though the receiver needs to reverse left and right inputs).

So far, we have said nothing about the function f . In fact, any f would work in the above procedures. But some choices of f yield much better security than others. The type of f used in DES is similar to that which we describe next. It is built up from a few components.

The first function is an expander. It takes an input of six bits and outputs eight bits. The one we use is given in

Figure 7.2.

Figure 7.2 The Expander Function

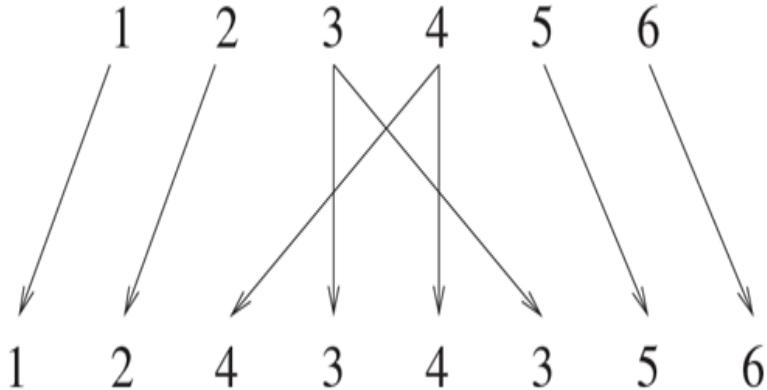


Figure 7.2 Full Alternative Text

This means that the first input bit yields the first output bit, the third input bit yields both the fourth and the sixth output bits, etc. For example, 011001 is expanded to 01010101.

The main components are called S-boxes. We use two:

$$S_1 \quad \begin{bmatrix} 101 & 010 & 001 & 110 & 011 & 100 & 111 & 000 \\ 001 & 100 & 110 & 010 & 000 & 111 & 101 & 011 \end{bmatrix}$$

$$S_2 \quad \begin{bmatrix} 100 & 000 & 110 & 101 & 111 & 001 & 011 & 010 \\ 101 & 011 & 000 & 111 & 110 & 010 & 001 & 100 \end{bmatrix}.$$

The input for an S-box has four bits. The first bit specifies which row will be used: 0 for the first row, 1 for the second. The other three bits represent a binary number that specifies the column: 000 for the first column, 001 for the second, ..., 111 for the last column. The output for the S-box consists of the three bits in the specified location. For example, an input of 1010 for S_1 means we look at the second row, third column, which yields the output 110.

The key K consists of nine bits. The key K_i for the i th round of encryption is obtained by using eight bits of K , starting with the i th bit. For example, if $K = 010011001$, then $K_4 = 01100101$ (after five bits, we reached the end of K , so the last two bits were obtained from the beginning of K).

We can now describe $f(R_{i-1}, K_i)$. The input R_{i-1} consists of six bits. The expander function is used to expand it to eight bits. The result is XORed with K_i to produce another eight-bit number. The first four bits are sent to S_1 , and the last four bits are sent to S_2 . Each S-box outputs three bits, which are concatenated to form a six-bit number. This is $f(R_{i-1}, K_i)$. We present this in Figure 7.3.

Figure 7.3 The Function

$$f(R_{i-1}, K_i)$$

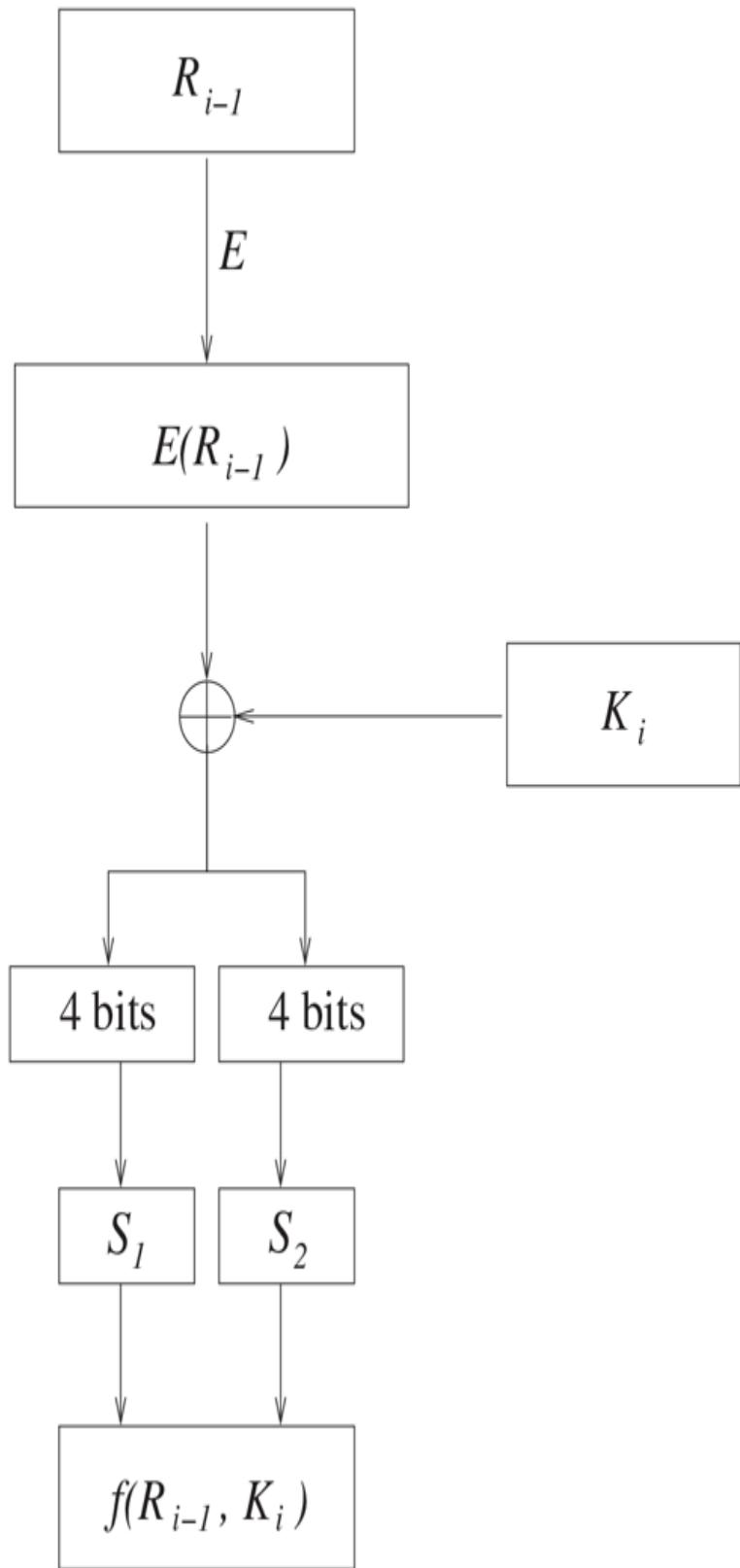


Figure 7.3 Full Alternative Text

For example, suppose $R_{i-1} = 100110$ and $K_i = 01100101$. We have

$$E(100110) \oplus K_i = 10101010 \oplus 01100101 = 11001111.$$

The first four bits are sent to S_1 and the last four bits are sent to S_2 . The second row, fifth column of S_1 contains 000. The second row, last column of S_2 contains 100.

Putting these outputs one after the other yields

$$f(R_{i-1}, K_i) = 000100.$$

We can now describe what happens in one round.

Suppose the input is

$$L_{i-1}R_{i-1} = 011100100110$$

and $K_i = 01100101$, as previously. This means that

$R_{i-1} = 100110$, as in the example just discussed.

Therefore, $f(R_{i-1}, K_i) = 000100$. This is XORed with

$L_{i-1} = 011100$ to yield $R_i = 011000$. Since

$L_i = R_{i-1}$, we obtain

$$L_iR_i = 100110011000.$$

The output becomes the input for the next round.

For work on this and another simplified DES algorithm and how they behave under multiple encryption, see [Konikoff-Toplosky].

7.3 Differential Cryptanalysis

This section is rather technical and can be skipped on a first reading.

Differential cryptanalysis was introduced by Biham and Shamir around 1990, though it was probably known much earlier to the designers of DES at IBM and NSA. The idea is to compare the differences in the ciphertexts for suitably chosen pairs of plaintexts and thereby deduce information about the key. Note that the difference of two strings of bits can be found by XORing them. Because the key is introduced by XORing with $E(R_{i-1})$, looking at the XOR of the inputs removes the effect of the key at this stage and hence removes some of the randomness introduced by the key. We'll see that this allows us to deduce information as to what the key could be.

7.3.1 Differential Cryptanalysis for Three Rounds

We eventually want to describe how to attack the above system when it uses four rounds, but we need to start by analyzing three rounds. Therefore, we temporarily start with L_1R_1 instead of L_0R_0 .

The situation is now as follows. We have obtained access to a three-round encryption device that uses the preceding procedure. We know all the inner workings of the encryption algorithm such as the S-boxes, but we do not know the key. We want to find the key by a chosen

plaintext attack. We use various inputs $L_1 R_1$ and obtain outputs $L_4 R_4$.

We have

$$\begin{aligned} R_2 &= L_1 \oplus f(R_1, K_2) \\ L_3 &= R_2 = L_1 \oplus f(R_1, K_2) \\ R_4 &= L_3 \oplus f(R_3, K_4) = L_1 \oplus f(R_1, K_2) \oplus f(R_3, K_4). \end{aligned}$$

Suppose we have another message $L_1^* R_1^*$ with $R_1 = R_1^*$. For each i , let $R_i' = R_i \oplus R_i^*$ and $L_i' = L_i \oplus L_i^*$. Then $L_i' R_i'$ is the “difference” (or sum; we are working mod 2) of $L_i R_i$ and $L_i^* R_i^*$. The preceding calculation applied to $L_1^* R_1^*$ yields a formula for R_4^* . Since we have assumed that $R_1 = R_1^*$, we have

$f(R_1, K_2) = f(R_1^*, K_2)$. Therefore,
 $f(R_1, K_2) \oplus f(R_1^*, K_2) = 0$ and

$$R_4' = R_4 \oplus R_4^* = L_1' \oplus f(R_3, K_4) \oplus f(R_3^*, K_4).$$

This may be rearranged to

$$R_4' \oplus L_1' = f(R_3, K_4) \oplus f(R_3^*, K_4).$$

Finally, since $R_3 = L_4$ and $R_3^* = L_4^*$, we obtain

$$R_4' \oplus L_1' = f(L_4, K_4) \oplus f(L_4^*, K_4).$$

Note that if we know the input XOR, namely $L_1' R_1'$, and if we know the outputs $L_4 R_4$ and $L_4^* R_4^*$, then we know everything in this last equation except K_4 .

Now let's analyze the inputs to the S-boxes used to calculate $f(L_4, K_4)$ and $f(L_4^*, K_4)$. If we start with L_4 , we first expand and then XOR with K_4 to obtain $E(L_4) \oplus K_4$, which are the bits sent to S_1 and S_2 . Similarly, L_4^* yields $E(L_4^*) \oplus K_4$. The XOR of these is

$$E(L_4) \oplus E(L_4^*) = E(L_4 \oplus L_4^*) = E(L_4')$$

(the first equality follows easily from the bit-by-bit description of the expansion function). Therefore, we know

1. the XORs of the inputs to the two S-boxes (namely, the first four and the last four bits of $E(L'_4)$);
2. the XORs of the two outputs (namely, the first three and the last three bits of $R'_4 \oplus L'_1$).

Let's restrict our attention to S_1 . The analysis for S_2 will be similar. It is fairly fast to run through all pairs of four-bit inputs with a given XOR (there are only 16 of them) and see which ones give a desired output XOR. These can be computed once for all and stored in a table.

For example, suppose we have input XOR equal to 1011 and we are looking for output XOR equal to 100. We can run through the input pairs (1011, 0000), (1010, 0001), (1001, 0010), ..., each of which has XOR equal to 1011, and look at the output XORs. We find that the pairs (1010, 0001) and (0001, 1010) both produce output XORs 100. For example, 1010 means we look at the second row, third column of S_1 , which is 110. Moreover, 0001 means we look at the first row, second column, which is 010. The output XOR is therefore $110 \oplus 010 = 100$.

We know L_4 and L_4^* . For example, suppose $L_4 = 101110$ and $L_4^* = 000010$. Therefore, $E(L_4) = 10111110$ and $E(L_4^*) = 00000010$, so the inputs to S_1 are $1011 \oplus K_4^L$ and $0000 \oplus K_4^L$, where K_4^L denotes the left four bits of K_4 . If we know that the output XOR for S_1 is 100, then $(1011 \oplus K_4^L, 0000 \oplus K_4^L)$ must be one of the pairs on the list we just calculated, namely (1010, 0001) and (0001, 1010). This means that $K_4^L = 0001$ or 1010.

If we repeat this procedure a few more times, we should be able to eliminate one of the two choices for K_4 and hence determine four bits of K . Similarly, using S_2 , we find four more bits of K . We therefore know eight of the nine bits of K . The last bit can be found by trying both

possibilities and seeing which one produces the same encryptions as the machine we are attacking.

Here is a summary of the procedure (for notational convenience, we describe it with both S-boxes used simultaneously, though in the examples we work with the S-boxes separately):

1. Look at the list of pairs with input $\text{XOR} = E(L'_4)$ and output $\text{XOR} = R'_4 \oplus L'_1$.
2. The pair $(E(L_4) \oplus K_4, E(L'_4) \oplus K_4)$ is on this list.
3. Deduce the possibilities for K_4 .
4. Repeat until only one possibility for K_4 remains.

Example

We start with

$$L_1R_1 = 000111011011$$

and the machine encrypts in three rounds using the key $K = 001001101$, though we do not yet know K . We obtain (note that since we are starting with L_1R_1 , we start with the shifted key $K_2 = 01001101$)

$$L_4R_4 = 000011100101.$$

If we start with

$$L_1^*R_1^* = 101110011011$$

(note that $R_1 = R_1^*$), then

$$L_4^*R_4^* = 100100011000.$$

We have $E(L_4) = 00000011$ and $E(L'_4) = 10101000$. The inputs to S_1 have XOR equal to 1010 and the inputs to S_2 have XOR equal to 1011. The S-boxes have output XOR $R'_4 \oplus L'_1 = 111101 \oplus 101001 = 010100$, so the output XOR from S_1 is 010 and that from S_2 is 100.

For the pairs $(1001, 0011)$, $(0011, 1001)$, S_1 produces output XOR equal to 010. Since the first member of one of these pairs should be the left four bits of $E(L_4) \oplus K_4 = 0000 \oplus K_4$, the first four bits of K_4 are in $\{1001, 0011\}$. For the pairs $(1100, 0111)$, $(0111, 1100)$, S_2 produces output XOR equal to 100. Since the first member of one of these pairs should be the right four bits of $E(L_4) \oplus K_4 = 0011 \oplus K_4$, the last four bits of K_4 are in $\{1111, 0100\}$.

Now repeat (with the same machine and the same key K) and with

$$L_1 R_1 = 010111011011 \text{ and } L_1^* R_1^* = 101110011011.$$

A similar analysis shows that the first four bits of K_4 are in $\{0011, 1000\}$ and the last four bits are in $\{0100, 1011\}$. Combining this with the previous information, we see that the first four bits of K_4 are 0011 and the last four bits are 0100. Therefore, $K = 00 * 001101$ (recall that K_4 starts with the fourth bit of K).

It remains to find the third bit of K . If we use $K = 000001101$, it encrypts $L_1 R_1$ to 001011101010, which is not $L_4 R_4$, while $K = 001001101$ yields the correct encryption. Therefore, the key is $K = 001001101$.

7.3.2 Differential Cryptanalysis for Four Rounds

Suppose now that we have obtained access to a four-round device. Again, we know all the inner workings of the algorithm except the key, and we want to determine the key. The analysis we used for three rounds still

applies, but to extend it to four rounds we need to use more probabilistic techniques.

There is a weakness in the box S_1 . If we look at the 16 input pairs with XOR equal to 0011, we discover that 12 of them have output XOR equal to 011. Of course, we expect on the average that two pairs should yield a given output XOR, so the present case is rather extreme. A little variation is to be expected; we'll see that this large variation makes it easy to find the key.

There is a similar weakness in S_2 , though not quite as extreme. Among the 16 input pairs with XOR equal to 1100, there are eight with output XOR equal to 010.

Suppose now that we start with randomly chosen R_0 and R_0^* such that $R_0' = R_0 \oplus R_0^* = 001100$. This is expanded to $E(001100) = 00111100$. Therefore the input XOR for S_1 is 0011 and the input XOR for S_2 is 1100. With probability 12/16 the output XOR for S_1 will be 011, and with probability 8/16 the output XOR for S_2 will be 010. If we assume the outputs of the two S-boxes are independent, we see that the combined output XOR will be 011010 with probability $(12/16)(8/16) = 3/8$. Because the expansion function sends bits 3 and 4 to both S_1 and S_2 , the two boxes cannot be assumed to have independent outputs, but 3/8 should still be a reasonable estimate for what happens.

Now suppose we choose L_0 and L_0^* so that $L_0' = L_0 \oplus L_0^* = 011010$. Recall that in the encryption algorithm the output of the S-boxes is XORed with L_0 to obtain R_1 . Suppose the output XOR of the S-boxes is 011010. Then $R_1' = 011010 \oplus L_0' = 000000$. Since $R_1' = R_1 \oplus R_1^*$, it follows that $R_1 = R_1^*$.

Putting everything together, we see that if we start with two randomly chosen messages with XOR equal to

$L'_0 R'_0 = 011010001100$, then there is a probability of around 3/8 that $L'_1 R'_1 = 001100000000$.

Here's the strategy for finding the key. Try several randomly chosen pairs of inputs with XOR equal to 011010001100 . Look at the outputs $L_4 R_4$ and $L_4^* R_4^*$. Assume that $L'_1 R'_1 = 001100000000$. Then use three-round differential cryptanalysis with $L'_1 = 001100$ and the known outputs to deduce a set of possible keys K_4 . When $L'_1 R'_1 = 001100000000$, which should happen around 3/8 of the time, this list of keys will contain K_4 , along with some other random keys. The remaining 5/8 of the time, the list should contain random keys. Since there seems to be no reason that any incorrect key should appear frequently, the correct key K_4 will probably appear in the lists of keys more often than the other keys.

Here is an example. Suppose we are attacking a four-round device. We try one hundred random pairs of inputs $L_0 R_0$ and $L_0^* R_0^* = L_0 R_0 \oplus 011010001100$. The frequencies of possible keys we obtain are in the following table. We find it easier to look at the first four bits and the last four bits of K_4 separately.

First four bits	Frequency	First four bits	Frequency
0000	12	1000	33
0001	7	1001	40
0010	8	1010	35
0011	15	1011	35
0100	4	1100	59
0101	3	1101	32
0110	4	1110	28
0111	6	1111	39

Last four bits	Frequency	Last four bits	Frequency
0000	14	1000	8
0001	6	1001	16
0010	42	1010	8
0011	10	1011	18
0100	27	1100	8
0101	10	1101	23
0110	8	1110	6
0111	11	1111	17

7.3-1 Full Alternative Text

It is therefore likely that $K_4 = 11000010$. Therefore, the key K is $10^*110000$.

To determine the remaining bit, we proceed as before. We can compute that $oooooooooooo$ is encrypted to 100011001011 using $K = 101110000$ and is encrypted to 001011011010 using $K = 100110000$. If the machine we are attacking encrypts $oooooooooooo$ to 100011001011 , we conclude that the second key cannot be correct, so the correct key is probably $K = 101110000$.

The preceding attack can be extended to more rounds by extensions of these methods. It might be noticed that we could have obtained the key at least as quickly by simply running through all possibilities for the key. That is certainly true in this simple model. However, in more elaborate systems such as DES, differential cryptanalytic techniques are much more efficient than exhaustive

searching through all keys, at least until the number of rounds becomes fairly large. In particular, the reason that DES uses 16 rounds appears to be because differential cryptanalysis is more efficient than exhaustive search until 16 rounds are used.

There is another attack on DES, called **linear cryptanalysis**, that was developed by Mitsuru Matsui [Matsui]. The main ingredient is an approximation of DES by a linear function of the input bits. It is theoretically faster than an exhaustive search for the key and requires around 2^{43} plaintext–ciphertext pairs to find the key. It seems that the designers of DES had not anticipated linear cryptanalysis. For details of the method, see [Matsui].

7.4 DES

A block of plaintext consists of 64 bits. The key has 56 bits, but is expressed as a 64-bit string. The 8th, 16th, 24th, ..., bits are parity bits, arranged so that each block of eight bits has an odd number of 1s. This is for error detection purposes. The output of the encryption is a 64-bit ciphertext.

The DES algorithm, depicted in Figure 7.4, starts with a plaintext m of 64 bits, and consists of three stages:

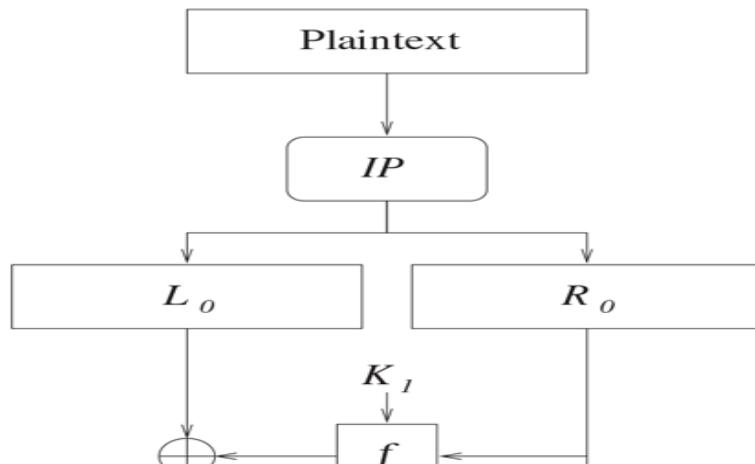
1. The bits of m are permuted by a fixed initial permutation to obtain $m_0 = IP(m)$. Write $m_0 = L_0R_0$, where L_0 is the first 32 bits of m_0 and R_0 is the last 32 bits.
2. For $1 \leq i \leq 16$, perform the following:

$$\begin{aligned}L_i &= R_{i-1} \\R_i &= L_{i-1} \oplus f(R_{i-1}, K_i),\end{aligned}$$

where K_i is a string of 48 bits obtained from the key K and f is a function to be described later.

3. Switch left and right to obtain $R_{16}L_{16}$, then apply the inverse of the initial permutation to get the ciphertext $c = IP^{-1}(R_{16}L_{16})$.

Figure 7.4 The DES Algorithm



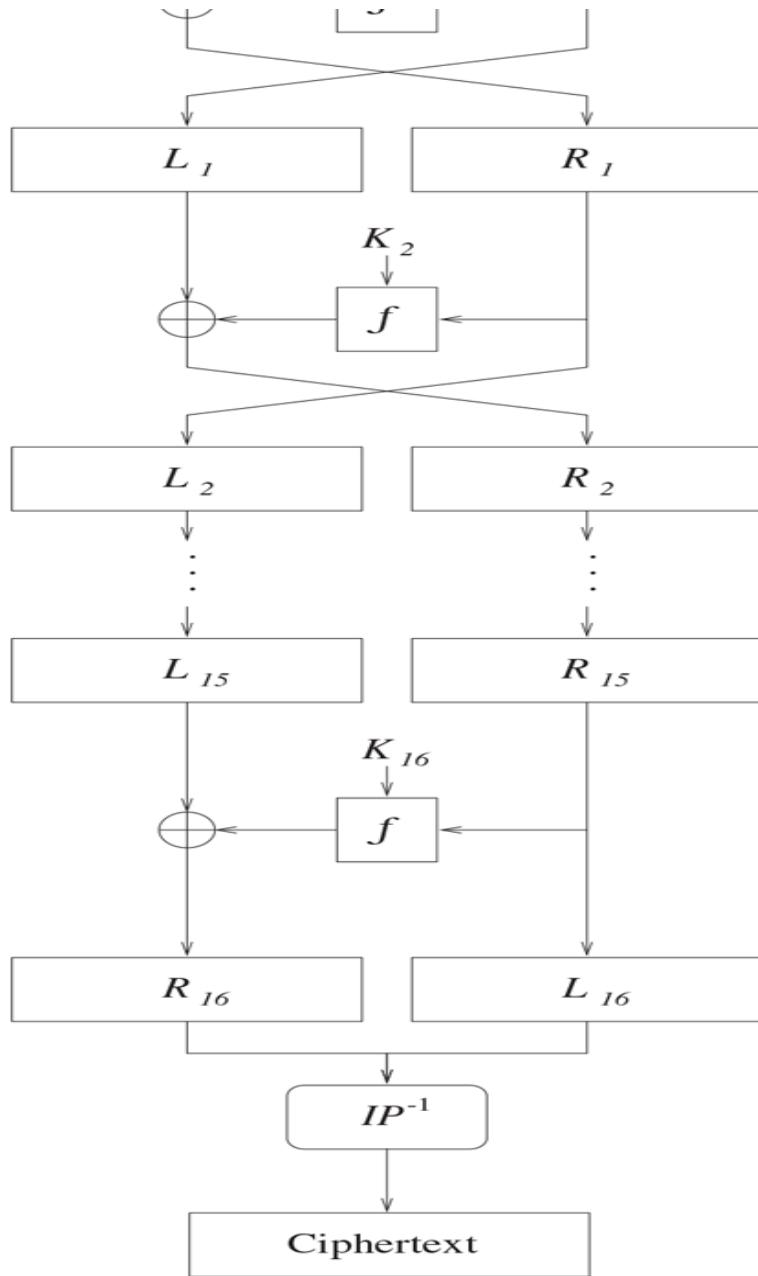


Figure 7.4 Full Alternative Text

Decryption is performed by exactly the same procedure, except that the keys K_1, \dots, K_{16} are used in reverse order. The reason this works is the same as for the simplified system described in [Section 7.2](#). Note that the left-right switch in step 3 of the DES algorithm means that we do not have to do the left-right switch that was needed for decryption in [Section 7.2](#).

We now describe the steps in more detail.

The initial permutation, which seems to have no cryptographic significance, but which was perhaps designed to make the algorithm load more efficiently into chips that were available in 1970s, can be described by the Initial Permutation table. This means that the 58th bit of m becomes the first bit of m_0 , the 50th bit of m becomes the second bit of m_0 , etc.

Initial Permutation															
58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

7.4-2 Full Alternative Text

The function $f(R, K_i)$, depicted in Figure 7.5, is described in several steps.

1. First, R is expanded to $E(R)$ by the following table.

Expansion Permutation															
32	1	2	3	4	5	4	5	6	7	8	9				
8	9	10	11	12	13	12	13	14	15	16	17				
16	17	18	19	20	21	20	21	22	23	24	25				
24	25	26	27	28	29	28	29	30	31	32	1				

7.4-3 Full Alternative Text

This means that the first bit of $E(R)$ is the 32nd bit of R , etc. Note that $E(R)$ has 48 bits.

2. Compute $E(R) \oplus K_i$, which has 48 bits, and write it as $B_1 B_2 \dots B_8$, where each B_j has six bits.
3. There are eight S-boxes S_1, \dots, S_8 , given on page 150. B_j is the input for S_j . Write $B_j = b_1 b_2 \dots b_6$. The row of the S-box is specified by $b_1 b_6$ while $b_2 b_3 b_4 b_5$ determines the column. For example, if $B_3 = 001001$, we look at the row 01, which is the second row (00 gives the first row) and column 0100, which is the 5th column (0100 represents 4 in binary; the first column is numbered 0, so the fifth is labeled 4). The entry in S_3 in this

location is 3, which is 3 in binary. Therefore, the output of S_3 is 0011 in this case. In this way, we obtain eight four-bit outputs C_1, C_2, \dots, C_8 .

4. The string $C_1C_2 \dots C_8$ is permuted according to the following table.

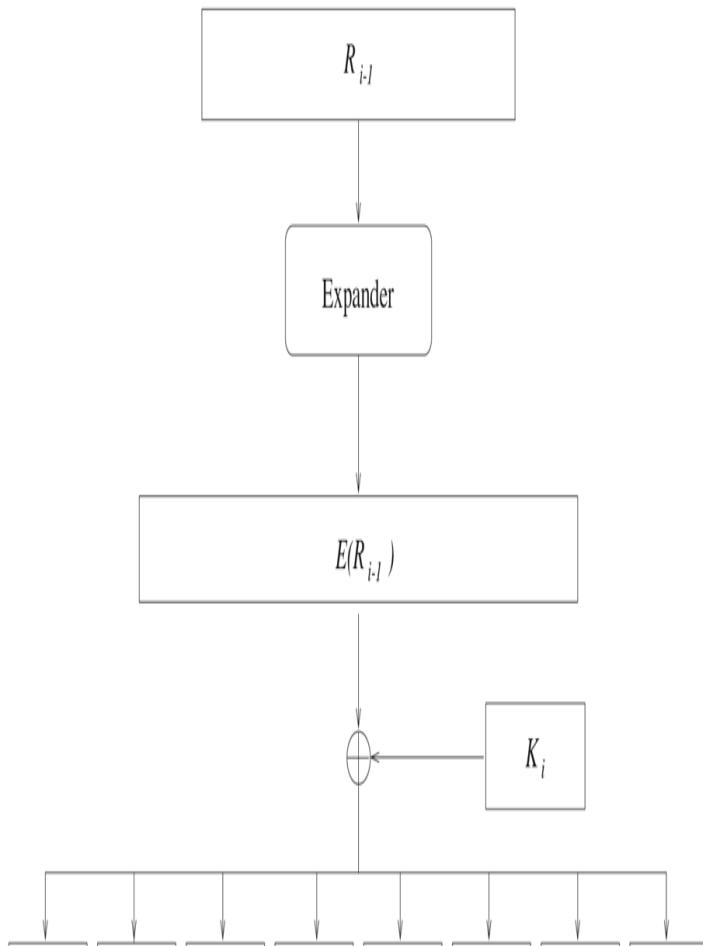
16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

7.4-4 Full Alternative Text

The resulting 32-bit string is $f(R, K_j)$.

Figure 7.5 The DES Function

$$f(R_{i-1}, K_i)$$



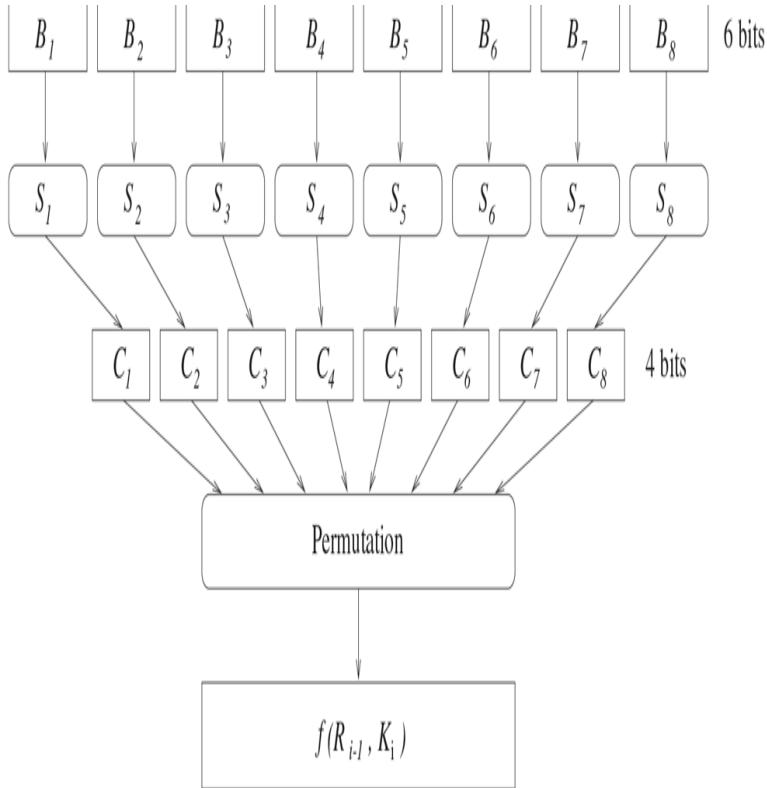


Figure 7.5 Full Alternative Text

Finally, we describe how to obtain K_1, \dots, K_{16} . Recall that we start with a 64-bit K .

1. The parity bits are discarded and the remaining bits are permuted by the following table.

Key Permutation															
57	49	41	33	25	17	9	1	58	50	42	34	26	18		
10	2	59	51	43	35	27	19	11	3	60	52	44	36		
63	55	47	39	31	23	15	7	62	54	46	38	30	22		
14	6	61	53	45	37	29	21	13	5	28	20	12	4		

7.4-5 Full Alternative Text

Write the result as C_0D_0 , where C_0 and D_0 have 28 bits.

2. For $1 \leq i \leq 16$, let $C_i = LS_i(C_{i-1})$ and $D_i = LS_i(D_{i-1})$. Here LS_i means shift the input one or two places to the left, according to the following table.

Number of Key Bits Shifted per Round																
Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	1	

7.4-6 Full Alternative Text

3. 48 bits are chosen from the 56-bit string $C_i D_i$ according to the following table. The output is K_i .

14	17	11	24	1	5	3	28	15	6	21	10					
23	19	12	4	26	8	16	7	27	20	13	2					
41	52	31	37	47	55	30	40	51	45	33	48					
44	49	39	56	34	53	46	42	50	36	29	32					

7.4-7 Full Alternative Text

It turns out that each bit of the key is used in approximately 14 of the 16 rounds.

A few remarks are in order. In a good cipher system, each bit of the ciphertext should depend on all bits of the plaintext. The expansion $E(R)$ is designed so that this will happen in only a few rounds. The purpose of the initial permutation is not completely clear. It has no cryptographic purpose. The S-boxes are the heart of the algorithm and provide the security. Their design was somewhat of a mystery until IBM published the following criteria in the early 1990s (for details, see [Coppersmith1]).

1. Each S-box has six input bits and four output bits. This was the largest that could be put on one chip in 1974.
2. The outputs of the S-boxes should not be close to being linear functions of the inputs (linearity would have made the system much easier to analyze).
3. Each row of an S-box contains all numbers from 0 to 15.
4. If two inputs to an S-box differ by one bit, the outputs must differ by at least two bits.

5. If two inputs to an S -box differ in exactly the middle two bits, then the outputs differ in at least two bits.
6. If two inputs to an S -box differ in their first two bits but have the same last two bits, the outputs must be unequal.
7. There are 32 pairs of inputs having a given XOR. For each of these pairs, compute the XOR of the outputs. No more than eight of these output XORs should be the same. This is clearly to avoid an attack via differential cryptanalysis.
8. A criterion similar to (7), but involving three S -boxes.

In the early 1970s, it took several months of searching for a computer to find appropriate S -boxes. Now, such a search could be completed in a very short time.

7.4.1 DES Is Not a Group

One possible way of effectively increasing the key size of DES is to double encrypt. Choose keys K_1 and K_2 and encrypt a plaintext P by $E_{K_2}(E_{K_1}(P))$. Does this increase the security?

S-Boxes

S-box 1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-box 2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S-box 3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S-box 4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S-box 5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S-box 6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S-box 7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S-box 8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

[Full Alternative Text](#)

Meet-in-the-middle attacks on cryptosystems are discussed in [Section 6.5](#). It is pointed out that, if an attacker has sufficient memory, double encryption provides little extra protection. Moreover, if a cryptosystem is such that double encryption is equivalent to a single encryption, then there is no additional security obtained by double encryption.

In addition, if double encryption is equivalent to single encryption, then the (single encryption) cryptosystem is much less secure than one might guess initially (see [Exercise 3 in Chapter 12](#)). If this were true for DES, for example, then an exhaustive search through all 2^{56} keys

could be replaced by a search of length around 2^{28} , which would be quite easy to do.

For affine ciphers ([Section 2.2](#)) and for RSA ([Chapter 9](#)), double encrypting with two keys K_1 and K_2 is equivalent to encrypting with a third key K_3 . Is the same true for DES? Namely, is there a key K_3 such that $E_{K_3} = E_{K_2}E_{K_1}$? This question is often rephrased in the equivalent form “Is DES a group?” (The reader who is unfamiliar with group theory can ask “Is DES closed under composition?”)

Fortunately, it turns out that DES is not a group. We sketch the proof. For more details, see [[Campbell-Wiener](#)]. Let E_0 represent encryption with the key consisting entirely of 0s and let E_1 represent encryption with the key consisting entirely of 1s. These keys are weak for cryptographic purposes (see [Exercise 5](#)). Moreover, D. Coppersmith found that applying $E_1 \circ E_0$ repeatedly to certain plaintexts yielded the original plaintext after around 2^{32} iterations. A sequence of encryptions (for some plaintext P)

$$E_1E_0(P), E_1E_0(E_1E_0(P)), E_1E_0(E_1E_0(E_1E_0(P))), \dots, (E_1E_0)^n(P) = P,$$

where n is the smallest positive integer such that $(E_1E_0)^n(P) = P$, is called a cycle of length n .

Lemma

If m is the smallest positive integer such that $(E_1E_0)^m(P) = P$ for all P , and n is the length of a cycle (so $(E_1E_0)^n(P_0) = P_0$ for a particular P_0), then n divides m .

Proof. Divide n into m , with remainder r . This means that $m = nq + r$ for some integer q , and $0 \leq r < n$. Since $(E_1E_0)^n(P_0) = P_0$, encrypting q times with $(E_1E_0)^n$ leaves P_0 unchanged. Therefore,

$$P_0 = (E_1 E_0)^m(P_0) = (E_1 E_0)^r (E_1 E_0)^{nq}(P_0) = (E_1 E_0)^r(P_0).$$

Since n is the smallest positive integer such that $(E_1 E_0)^n(P_0) = P_0$, and $0 \leq r < n$, we must have $r = 0$. This means that $m = nq$, so n divides m .

Suppose now that DES is closed under composition.

Then $E_1 E_0 = E_K$ for some key K . Moreover,

E_K^2, E_K^3, \dots are also represented by DES keys. Since there are only 2^{56} possible keys, we must have

$E_K^j = E_K^i$ for some integers i, j with $0 \leq i < j \leq 2^{56}$ (otherwise we would have $2^{56} + 1$ distinct encryption keys). Decrypt i times: $E_K^{j-i} = D_K^i E_K^j = D_K^i E_K^i$, which is the identity map. Since $0 < j - i \leq 2^{56}$, the smallest positive integer m such that E_K^m is the identity map also satisfies $m \leq 2^{56}$.

Coppersmith found the lengths of the cycles for 33 plaintexts P_0 . By the lemma, m is a multiple of these cycle lengths. Therefore, m is greater than or equal to the least common multiple of these cycle lengths, which turned out to be around 10^{277} . But if DES is closed under composition, we showed that $m \leq 2^{56}$. Therefore, DES is not closed under composition.

7.5 Breaking DES

DES was the standard cryptographic system for the last 20 years of the twentieth century, but, in the latter half of this period, DES was showing signs of age. In this section we discuss the breaking of DES.

From 1975 onward, there were questions regarding the strength of DES. Many in the academic community complained about the size of the DES keys, claiming that a 56-bit key was insufficient for security. In fact, a few months after the NBS release of DES, Whitfield Diffie and Martin Hellman published a paper titled “Exhaustive cryptanalysis of the NBS Data Encryption Standard” [Diffie-Hellman2] in which they estimated that a machine could be built for \$20 million (in 1977 dollars) that would crack DES in roughly a day. This machine’s purpose was specifically to attack DES, which is a point that we will come back to later.

In 1987 DES came under its second five-year review. At this time, NBS asked for suggestions whether to accept the standard for another period, to modify the standard, or to dissolve the standard altogether. The discussions regarding DES saw NSA opposing the recertification of DES. The NBS argued at that time that DES was beginning to show signs of weakness, given the current level of computing power, and proposed doing away with DES entirely and replacing it with a set of NSA-designed algorithms whose inner workings would be known only to NSA and be well protected from reverse engineering techniques. This proposal was turned down, partially due to the fact that several key American industries would be left unprotected while replacement algorithms were put in place. In the end, DES was reapproved as a standard,

yet in the process it was acknowledged that DES was showing signs of weakness.

Five years later, after NBS had been renamed NIST, the next five-year review came around. Despite the weaknesses mentioned in 1987 and the technology advances that had taken place in five years, NIST recertified the DES algorithm in 1992.

In 1993, Michael Wiener, a researcher at Bell-Northern Research, proposed and designed a device that would attack DES more efficiently than ever before. The idea was to use the already well-developed switching technology available to the telephone industry.

The year 1996 saw the formulation of three basic approaches for attacking symmetric ciphers such as DES. The first method was to do distributive computation across a vast collection of machines. This had the advantage that it was relatively cheap, and the cost that was involved could be easily distributed over many people. Another approach was to design custom architecture (such as Michael Wiener's idea) for attacking DES. This promised to be more effective, yet also more expensive, and could be considered as the high-end approach. The middle-of-the-line approach involved programmable logic arrays and has received the least attention to date.

In all three of these cases, the most popular approach to attacking DES was to perform an exhaustive search of the keyspace. For DES this seemed to be reasonable since, as mentioned earlier, more complicated cryptanalytic techniques had failed to show significant improvement over exhaustive search.

The distributive computing approach to breaking DES became very popular, especially with the growing popularity of the Internet. In 1997 the RSA Data Security company issued a challenge to find the key and crack a

DES encrypted message. Whoever cracked the message would win a \$10,000 prize. Only five months after the announcement of the 1997 DES Challenge, Rocke Verser submitted the winning DES key. What is important about this is that it represents an example where the distributive computing approach had successfully attacked DES. Rocke Verser had implemented a program where thousands of computers spread over the Internet had managed to crack the DES cipher. People volunteered time on their personal (and corporate) machines, running Verser's program under the agreement that Verser would split the winnings 60% to 40% with the owner of the computer that actually found the key. The key was finally found by Michael Sanders. Roughly 25% of the DES keyspace had been searched by that time. The DES Challenge phrase decrypted to "Strong cryptography makes the world a safer place."

In the following year, RSA Data Security issued DES Challenge II. This time the correct key was found by Distributed Computing Technologies, and the message decrypted to "Many hands make light work." The key was found after searching roughly 85% of the possible keys and was done in 39 days. The fact that the winner of the second challenge searched more of the keyspace and performed the task quicker than the first task shows the dramatic effect that a year of advancement in technology can have on cryptanalysis.

In the summer of 1998 the Electronic Frontier Foundation (EFF) developed a project called DES Cracker whose purpose was to reveal the vulnerability of the DES algorithm when confronted with a specialized architecture. The DES Cracker project was founded on a simple principle: The average computer is ill suited for the task of cracking DES. This is a reasonable statement since ordinary computers, by their very nature, are multipurpose machines that are designed to handle generic tasks such as running an operating system or

even playing a computer game or two. What the EFF team proposed to do was build specialized hardware that would take advantage of the parallelizable nature of the exhaustive search. The team had a budget of \$200,000.

We now describe briefly the architecture that the EFF team's research produced. For more information regarding the EFF Cracker as well as the other tasks their cracker was designed to handle, see [Gilmore].

The EFF DES Cracker consisted of basically three main parts: a personal computer, software, and a large collection of specialized chips. The computer was connected to the array of chips and the software oversaw the tasking of each chip. For the most part, the software didn't interact much with the hardware; it just gave the chips the necessary information to start processing and waited until the chips returned candidate keys. In this sense, the hardware efficiently eliminated a large number of invalid keys and only returned keys that were potentially promising. The software then processed each of the promising candidate keys on its own, checking to see if one of the promising keys was in fact the actual key.

The DES Cracker took a 128-bit (16-byte) sample of ciphertext and broke it into two 64-bit (8-byte) blocks of text. Each chip in the EFF DES Cracker consisted of 24 search units. A search unit was a subset of a chip whose task was to take a key and two 64-bit blocks of ciphertext and attempt to decrypt the first 64-bit block using the key. If the "decrypted" ciphertext looked interesting, then the search unit decrypted the second block and checked to see if that "decrypted" ciphertext was also interesting. If both decrypted texts were interesting then the search unit told the software that the key it checked was promising. If, when the first 64-bit block of ciphertext was decrypted, the decrypted text did not seem interesting enough, then the search unit

incremented its key by 1 to form a new key. It then tried this new key, again checking to see if the result was interesting, and proceeded this way as it searched through its allotted region of keyspace.

How did the EFF team define an “interesting” decrypted text? First they assumed that the plaintext satisfied some basic assumption, for example that it was written using letters, numbers, and punctuation. Since the data they were decrypting was text, they knew each byte corresponded to an eight-bit character. Of the 256 possible values that an eight-bit character type represented, only 69 characters were interesting (the uppercase and lowercase alphabet, the numbers, the space, and a few punctuation marks). For a byte to be considered interesting, it had to contain one of these 69 characters, and hence had a $69/256$ chance of being interesting. Approximating this ratio to $1/4$, and assuming that the decrypted bytes are in fact independent, we see that the chance that an 8-byte block of decrypted text was interesting is $1/4^8 = 1/65536$. Thus only $1/65536$ of the keys it examined were considered promising.

This was not enough of a reduction. The software would still spend too much time searching false candidates. In order to narrow down the field of promising key candidates even further, it was necessary to use the second 8-byte block of text. This block was decrypted to see if the result was interesting. Assuming independence between the blocks, we get that only $1/4^{16} = 1/65536^2$ of the keys could be considered promising. This significantly reduced the amount of keyspace that the software had to examine.

Each chip consisted of 24 search units, and each search unit was given its own region of the keyspace that it was responsible for searching. A single 40-MHz chip would have taken roughly 38 years to search the entire

keyspace. To reduce further the amount of time needed to process the keys, the EFF team used 64 chips on a single circuit board, then 12 boards to each chassis, and finally two chassis were connected to the personal computer that oversaw the communication with the software.

The end result was that the DES Cracker consisted of about 1500 chips and could crack DES in roughly 4.5 days on average. The DES Cracker was by no means an optimum model for cracking DES. In particular, each of the chips that it used ran at 40 MHz, which is slow by modern standards. Newer models could certainly be produced in the future that employ chips running at much faster clock cycles.

This development strongly indicated the need to replace DES. There were two main approaches to achieving increased security. The first used DES multiple times and led to the popular method called Triple DES or 3DES. Multiple encryption for block ciphers is discussed in [Section 6.4](#).

The second approach was to find a new system that employs a larger key size than 56 bits. This led to AES, which is discussed in [Chapter 8](#).

7.6 Password Security

When you log in to a computer and enter your password, the computer checks that your password belongs to you and then grants access. However, it would be quite dangerous to store the passwords in a file in the computer. Someone who obtains that file would then be able to open anyone's account. Making the file available only to the computer administrator might be one solution; but what happens if the administrator makes a copy of the file shortly before changing jobs? The solution is to encrypt the passwords before storing them.

Let $f(x)$ be a **one-way function**. This means that it is easy to compute $f(x)$, but it is very difficult to solve $y = f(x)$ for x . A password x can then be stored as $f(x)$, along with the user's name. When the user logs in, and enters the password x , the computer calculates $f(x)$ and checks that it matches the value of $f(x)$ corresponding to that user. An intruder who obtains the password file will have only the value of $f(x)$ for each user. To log in to the account, the intruder needs to know x , which is hard to compute since $f(x)$ is a one-way function.

In many systems, the encrypted passwords are stored in a public file. Therefore, anyone with access to the system can obtain this file. Assume the function $f(x)$ is known. Then all the words in a dictionary, and various modifications of these words (writing them backward, for example) can be fed into $f(x)$. Comparing the results with the password file will often yield the passwords of several users.

This **dictionary attack** can be partially prevented by making the password file not publicly available, but there

is still the problem of the departing (or fired) computer administrator. Therefore, other ways of making the information more secure are also needed.

Here is another interesting problem. It might seem desirable that $f(x)$ can be computed very quickly. However, a slightly slower $f(x)$ can slow down a dictionary attack. But slowing down $f(x)$ too much could also cause problems. If $f(x)$ is designed to run in a tenth of a second on a very fast computer, it could take an unacceptable amount of time to log in on a slower computer. There doesn't seem to be a completely satisfactory way to resolve this.

One way to hinder a dictionary attack is with what is called **salt**. Each password is randomly padded with an additional 12 bits. These 12 bits are then used to modify the function $f(x)$. The result is stored in the password file, along with the user's name and the values of the 12-bit salt. When a user enters a password x , the computer finds the value of the salt for this user in the file, then uses it in the computation of the modified $f(x)$, which is compared with the value stored in the file.

When salt is used and the words in the dictionary are fed into $f(x)$, they need to be padded with each of the $2^{12} = 4096$ possible values of the salt. This slows down the computations considerably. Also, suppose an attacker stores the values of $f(x)$ for all the words in the dictionary. This could be done in anticipation of attacking several different password files. With salt, the storage requirements increase dramatically, since each word needs to be stored 4096 times.

The main purpose of salt is to stop attacks that aim at finding a random person's password. In particular, it makes the set of poorly chosen passwords somewhat more secure. Since many people use weak passwords, this is desirable. Salt does not slow down an attack

against an individual password (except by preventing use of over-the-counter DES chips; see below). If Eve wants to find Bob's password and has access to the password file, she finds the value of the salt used for Bob and tries a dictionary attack, for example, using only this value of salt corresponding to Bob. If Bob's password is not in the dictionary, this will fail, and Eve may have to resort to an exhaustive search of all possible passwords.

In many Unix password schemes, the one-way function was based on DES. The first eight characters of the password are converted to seven-bit ASCII (see [Section 4.1](#)). These 56 bits become a DES key. If the password is shorter than eight symbols, it is padded with zeros to obtain the 56 bits. The “plaintext” of all zeros is then encrypted using 25 rounds of DES with this key. The output is stored in the password file. The function

$$\text{password} \rightarrow \text{output}$$

is believed to be one-way. Namely, we know the “ciphertext,” which is the output, and the “plaintext,” which is all zeros. Finding the key, which is the password, amounts to a known plaintext attack on DES, which is generally assumed to be difficult.

In order to increase security, salt is added as follows. A random 12-bit number is generated as the salt. Recall that in DES, the expansion function E takes a 32-bit input R (the right side of the input for the round) and expands it to 48 bits $E(R)$. If the first bit of the salt is 1, the first and 25th bits of $E(R)$ are swapped. If the second bit of the salt is 1, the second and 26th bits of $E(R)$ are swapped. This continues through the 12th bit of the salt. If it is 1, the 12th and 36th bits of $E(R)$ are swapped. When a bit of the salt is 0, it causes no swap. If the salt is all zero, then no swaps occur and we are working with the usual DES. In this way, the salt means that 4096 variations of DES are possible.

One advantage of using salt to modify DES is that someone cannot use high-speed DES chips to compute the one-way function when performing a dictionary attack. Instead, a chip would need to be designed that tries all 4096 modifications of DES caused by the salt; otherwise the attack could be performed with software, which is much slower.

Salt in any password scheme is regarded by many as a temporary measure. As storage space increases and computer speed improves, a factor of 4096 quickly fades, so eventually a new system must be developed.

For more on password protocols, see [Section 12.6](#).

7.7 Exercises

1. Consider the following DES-like encryption method. Start with a message of $2n$ bits. Divide it into two blocks of length n (a left half and a right half): M_0M_1 . The key K consists of k bits, for some integer k . There is a function $f(K, M)$ that takes an input of k bits and n bits and gives an output of n bits. One round of encryption starts with a pair M_jM_{j+1} . The output is the pair $M_{j+1}M_{j+2}$, where

$$M_{j+2} = M_j \oplus f(K, M_{j+1}).$$

(\oplus means XOR, which is addition mod 2 on each bit). This is done for m rounds, so the ciphertext is M_mM_{m+1} .

1. If you have a machine that does the m -round encryption just described, how would you use the same machine to decrypt the ciphertext M_mM_{m+1} (using the same key K)? Prove that your decryption method works.
 2. Suppose K has n bits and $f(K, M) = K \oplus M$, and suppose the encryption process consists of $m = 2$ rounds. If you know only a ciphertext, can you deduce the plaintext and the key? If you know a ciphertext and the corresponding plaintext, can you deduce the key? Justify your answers.
 3. Suppose K has n bits and $f(K, M) = K \oplus M$, and suppose the encryption process consists of $m = 3$ rounds. Why is this system not secure?
2. Bud gets a budget 2-round Feistel system. It uses a 32-bit L , a 32-bit R , and a 32-bit key K . The function is $f(R, K) = R \oplus K$, with the same key for each round. Moreover, to avoid transmission errors, he always uses a 32-bit message M and lets $L_0 = R_0 = M$. Eve does not know Bud's key, but she obtains the ciphertext for one of Bud's encryptions. Describe how Eve can obtain the plaintext M and the key K .
3. As described in [Section 7.6](#), a way of storing passwords on a computer is to use DES with the password as the key to encrypt a fixed plaintext (usually `000...0`). The ciphertext is then stored in the file. When you log in, the procedure is repeated and the ciphertexts are compared. Why is this method more secure than the similar-sounding method of using the password as the plaintext and using a fixed key (for example, `000...0`)?

4. Nelson produces budget encryption machines for people who cannot afford a full-scale version of DES. The encryption consists of one round of a Feistel system. The plaintext has 64 bits and is divided into a left half L and a right half R . The encryption uses a function $f(R)$ that takes an input string of 32 bits and outputs a string of 32 bits. (There is no key; anyone naive enough to buy this system should not be trusted to choose a key.) The left half of the ciphertext is $C_0 = R$ and the right half is $C_1 = L \oplus f(R)$. Suppose Alice uses one of these machines to encrypt and send a message to Bob. Bob has an identical machine. How does he use the machine to decrypt the ciphertext he receives? Show that this decryption works (do not quote results about Feistel systems; you are essentially justifying that a special case works).

5.
 1. Let $K = 111 \dots 111$ be the DES key consisting of all 1s. Show that if $E_K(P) = C$, then $E_K(C) = P$, so encryption twice with this key returns the plaintext. (Hint: The round keys are sampled from K . Decryption uses these keys in reverse order.)
 2. Find another key with the same property as K in part (a).

6. Alice uses quadruple DES encryption. To save time, she chooses two keys, K_1 and K_2 , and encrypts via $c = E_{K_1}(E_{K_1}(E_{K_2}(E_{K_2}(m))))$. One day, Alice chooses K_1 to be the key of all 1s and K_2 to be the key of all 0s. Eve is planning to do a meet-in-the-middle attack, but after examining a few plaintext–ciphertext pairs, she decides that she does not need to carry out this attack. Why? (Hint: Look at Exercise 5.)

7. For a string of bits S , let \bar{S} denote the complementary string obtained by changing all the 1s to 0s and all the 0s to 1s (equivalently, $\bar{S} = S \oplus 11111 \dots$). Show that if the DES key K encrypts P to C , then K encrypts \bar{P} to \bar{C} . (Hint: This has nothing to do with the structure of the S-boxes. To do the problem, just work through the encryption algorithm and show that the input to the S-boxes is the same for both encryptions. A key point is that the expansion of C is the complementary string for the expansion of \bar{C} .)

8. Suppose we modify the Feistel setup as follows. Divide the plaintext into three equal blocks: L_0, M_0, R_0 . Let the key for the i th round be K_i and let f be some function that produces the appropriate size output. The i th round of encryption is given by

$$L_i = R_{i-1}, M_i = L_{i-1}, R_i = f(K_i, R_{i-1}) \oplus M_{i-1}.$$

This continues for n rounds. Consider the decryption algorithm that starts with the ciphertext A_n, B_n, C_n and uses the algorithm

$$A_{i-1} = B_i, B_{i-1} = f(K_i, A_i) \oplus C_i, C_{i-1} = A_i.$$

This continues for n rounds, down to A_0, B_0, C_0 . Show that $A_i = L_i, B_i = M_i, C_i = R_i$ for all i , so that the decryption algorithm returns the plaintext. (Remark: Note that the decryption algorithm is similar to the encryption algorithm, but cannot be implemented on the same machine as easily as in the case of the Feistel system.)

9. Suppose $E_K(M)$ is the DES encryption of a message M using the key K . We showed in [Exercise 7](#) that DES has the complementation property, namely that if $y = E_K(M)$ then $y = E_{\bar{K}}(\bar{M})$, where \bar{M} is the bit complement of M . That is, the bitwise complement of the key and the plaintext result in the bitwise complement of the DES ciphertext. Explain how an adversary can use this property in a brute force, chosen plaintext attack to reduce the expected number of keys that would be tried from 2^{55} to 2^{54} . (Hint: Consider a chosen plaintext set of (M_1, C_1) and (M_1, C_2)).

7.8 Computer Problems

1. (For those who are comfortable with programming)
 1. Write a program that performs one round of the simplified DES-type algorithm presented in [Section 7.2](#).
 2. Create a sample input bitstring, and a random key. Calculate the corresponding ciphertext when you perform one round, two rounds, three rounds, and four rounds of the Feistel structure using your implementation. Verify that the decryption procedure works in each case.
 3. Let $E_K(M)$ denote four-round encryption using the key K . By trying all 2^9 keys, show that there are no weak keys for this simplified DES-type algorithm. Recall that a weak key is one such that when we encrypt a plaintext twice we get back the plaintext. That is, a weak key K satisfies $E_K(E_K(M)) = M$ for every possible M .
(Note: For each key K , you need to find some M such that $E_K(E_K(M)) \neq M$.)
 4. Suppose you modify the encryption algorithm $E_K(M)$ to create a new encryption algorithm $E'_K(M)$ by swapping the left and right halves after the four Feistel rounds. Are there any weak keys for this algorithm?
2. Using your implementation of $E_K(M)$ from [Computer Problem 1\(b\)](#), implement the CBC mode of operation for this simplified DES-type algorithm.
 1. Create a plaintext message consisting of 48 bits, and show how it encrypts and decrypts using CBC.
 2. Suppose that you have two plaintexts that differ in the 14th bit. Show the effect that this has on the corresponding ciphertexts.

Chapter 8 The Advanced Encryption Standard: Rijndael

In 1997, the National Institute of Standards and Technology put out a call for candidates to replace DES. Among the requirements were that the new algorithm should allow key sizes of 128, 192, and 256 bits, it should operate on blocks of 128 input bits, and it should work on a variety of different hardware, for example, eight-bit processors that could be used in smart cards and the 32-bit architecture commonly used in personal computers. Speed and cryptographic strength were also important considerations. In 1998, the cryptographic community was asked to comment on 15 candidate algorithms. Five finalists were chosen: MARS (from IBM), RC6 (from RSA Laboratories), Rijndael (from Joan Daemen and Vincent Rijmen), Serpent (from Ross Anderson, Eli Biham, and Lars Knudsen), and Twofish (from Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson). Eventually, Rijndael was chosen as the Advanced Encryption Standard. The other four algorithms are also very strong, and it is likely that they will be used in many future cryptosystems.

As with other block ciphers, Rijndael can be used in several modes, for example, ECB, CBC, CFB, OFB, and CTR (see [Section 6.3](#)).

Before proceeding to the algorithm, we answer a very basic question: How do you pronounce Rijndael? We quote from their Web page:

If you're Dutch, Flemish, Indonesian, Surinamer or South-African, it's pronounced like you think it should be. Otherwise, you could pronounce it like "Reign Dahl,"

“Rain Doll,” “Rhine Dahl”. We’re not picky. As long as you make it sound different from “Region Deal.”

8.1 The Basic Algorithm

Rijndael is designed for use with keys of lengths 128, 192, and 256 bits. For simplicity, we'll restrict to 128 bits.

First, we give a brief outline of the algorithm, then describe the various components in more detail.

The algorithm consists of 10 rounds (when the key has 192 bits, 12 rounds are used, and when the key has 256 bits, 14 rounds are used). Each round has a round key, derived from the original key. There is also a 0th round key, which is the original key. A round starts with an input of 128 bits and produces an output of 128 bits.

There are four basic steps, called **layers**, that are used to form the rounds:

1. The SubBytes Transformation (SB): This nonlinear layer is for resistance to differential and linear cryptanalysis attacks.
2. The ShiftRows Transformation (SR): This linear mixing step causes diffusion of the bits over multiple rounds.
3. The MixColumns Transformation (MC): This layer has a purpose similar to ShiftRows.
4. AddRoundKey (ARK): The round key is *XORED* with the result of the above layer.

A round is then

$\rightarrow \boxed{\text{SubBytes}} \rightarrow \boxed{\text{ShiftRows}} \rightarrow \boxed{\text{MixColumns}} \rightarrow \boxed{\text{AddRoundKey}} \rightarrow .$

Putting everything together, we obtain the following (see also [Figure 8.1](#)):

Figure 8.1The AES-Rijndael Algorithm

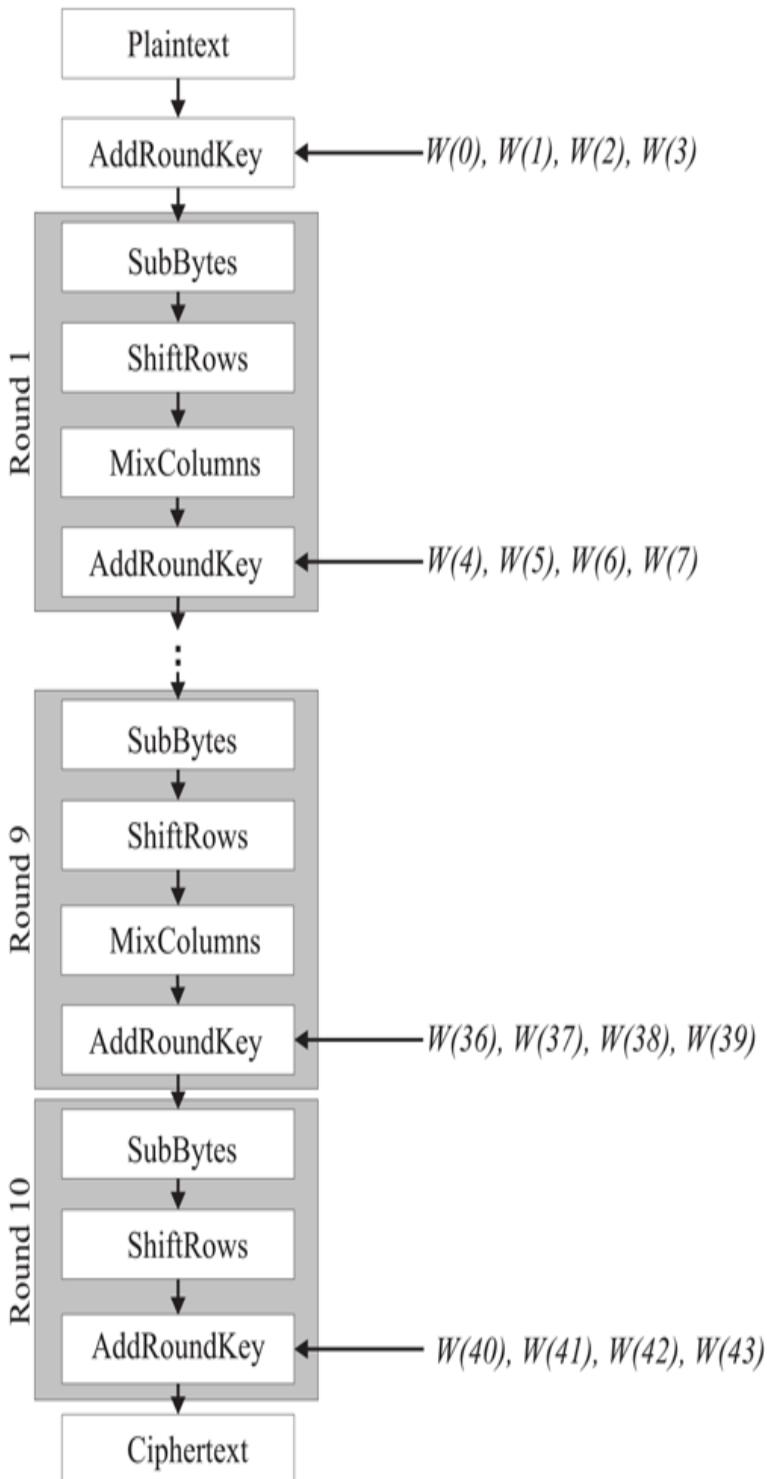


Figure 8.1 Full Alternative Text

Rijndael Encryption

1. ARK, using the 0th round key.
2. Nine rounds of SB, SR, MC, ARK, using round keys 1 to 9.
3. A final round: SB, SR, ARK, using the 10th round key.

8.1-1 Full Alternative Text

The final round uses the SubBytes, ShiftRows, and AddRoundKey steps but omits MixColumns (this omission will be explained in the decryption section).

The 128-bit output is the ciphertext block.

8.2 The Layers

We now describe the steps in more detail. The 128 input bits are grouped into 16 bytes of eight bits each, call them

$$a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, a_{1,1}, \dots, a_{3,3}.$$

These are arranged into a 4×4 matrix

$$\begin{matrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{matrix}.$$

In the following, we'll occasionally need to work with the finite field $GF(2^8)$. This is covered in [Section 3.11](#).

However, for the present purposes, we only need the following facts. The elements of $GF(2^8)$ are bytes, which consist of eight bits. They can be added by *XOR*. They can also be multiplied in a certain way (i.e., the product of two bytes is again a byte), but this process is more complicated. Each byte b except the zero byte has a multiplicative inverse; that is, there is a byte b' such that $b \cdot b' = 00000001$. Since we can do arithmetic operations on bytes, we can work with matrices whose entries are bytes.

As a technical point, we note that the model of $GF(2^8)$ depends on a choice of irreducible polynomial of degree 8. The choice for Rijndael is $X^8 + X^4 + X^3 + X + 1$. This is also the polynomial used in the examples in [Section 3.11](#). Other choices for this polynomial would presumably give equally good algorithms.

8.2.1 The SubBytes Transformation

In this step, each of the bytes in the matrix is changed to another byte by Table 8.1, called the S-box.

Table 8.1 S-Box for Rijndael

S-Box															
99	124	119	123	242	107	111	197	48	1	103	43	254	215	171	118
202	130	201	125	250	89	71	240	173	212	162	175	156	164	114	192
183	253	147	38	54	63	247	204	52	165	229	241	113	216	49	21
4	199	35	195	24	150	5	154	7	18	128	226	235	39	178	117
9	131	44	26	27	110	90	160	82	59	214	179	41	227	47	132
83	209	0	237	32	252	177	91	106	203	190	57	74	76	88	207
208	239	170	251	67	77	51	133	69	249	2	127	80	60	159	168
81	163	64	143	146	157	56	245	188	182	218	33	16	255	243	210
205	12	19	236	95	151	68	23	196	167	126	61	100	93	25	115
96	129	79	220	34	42	144	136	70	238	184	20	222	94	11	219
224	50	58	10	73	6	36	92	194	211	172	98	145	149	228	121
231	200	55	109	141	213	78	169	108	86	244	234	101	122	174	8
186	120	37	46	28	166	180	198	232	221	116	31	75	189	139	138
112	62	181	102	72	3	246	14	97	53	87	185	134	193	29	158
225	248	152	17	105	217	142	148	155	30	135	233	206	85	40	223
140	161	137	13	191	230	66	104	65	153	45	15	176	84	187	22

Table 8.1 Full Alternative Text

Write a byte as eight bits: $abcdefgh$. Look for the entry in the $abcd$ row and $efgh$ column (the rows and columns are numbered from 0 to 15). This entry, when converted to binary, is the output. For example, if the input byte is 10001011, we look in row 8 (the ninth row) and column 11 (the twelfth column). The entry is 61, which is 111101 in binary. This is the output of the S-box.

The output of SubBytes is again a 4×4 matrix of bytes, let's call it

$$\begin{array}{cccc}
 b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\
 b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\
 b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\
 b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3}
 \end{array} .$$

8.2.2 The ShiftRows Transformation

The four rows of the matrix are shifted cyclically to the left by offsets of 0, 1, 2, and 3, to obtain

$$\begin{array}{cccc} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{array} = \begin{array}{cccc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{array} .$$

8.2.3 The MixColumns Transformation

Regard a byte as an element of $GF(2^8)$, as in [Section 3.11](#). Then the output of the ShiftRows step is a 4×4 matrix $(c_{i,j})$ with entries in $GF(2^8)$. Multiply this by a matrix, again with entries in $GF(2^8)$, to produce the output $(d_{i,j})$, as follows:

$$\begin{array}{cccc} 00000010 & 00000011 & 00000001 & 00000001 \\ 00000001 & 00000010 & 00000011 & 00000001 \\ 00000001 & 00000001 & 00000010 & 00000011 \\ 00000011 & 00000001 & 00000001 & 00000010 \end{array} \begin{array}{cccc} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{array} = \begin{array}{cccc} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{array} .$$

8.2.4 The RoundKey Addition

The round key, derived from the key in a way we'll describe later, consists of 128 bits, which are arranged in a 4×4 matrix $(k_{i,j})$ consisting of bytes. This is XORed with the output of the MixColumns step:

$$\begin{array}{cccc} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{array} \oplus \begin{array}{cccc} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{array} = \begin{array}{cccc} e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \end{array}.$$

This is the final output of the round.

8.2.5 The Key Schedule

The original key consists of 128 bits, which are arranged into a 4×4 matrix of bytes. This matrix is expanded by adjoining 40 more columns, as follows. Label the first four columns $W(0), W(1), W(2), W(3)$. The new columns are generated recursively. Suppose columns up through $W(i-1)$ have been defined. If i is not a multiple of 4, then

$$W(i) = W(i-4) \oplus W(i-1).$$

If i is a multiple of 4, then

$$W(i) = W(i-4) \oplus T(W(i-1)),$$

where $T(W(i-1))$ is the transformation of $W(i-1)$ obtained as follows. Let the elements of the column $W(i-1)$ be a, b, c, d . Shift these cyclically to obtain b, c, d, a . Now replace each of these bytes with the corresponding element in the S-box from the SubBytes step, to get 4 bytes e, f, g, h . Finally, compute the round constant

$$r(i) = 00000010^{(i-4)/4}$$

in $GF(2^8)$ (recall that we are in the case where i is a multiple of 4). Then $T(W(i - 1))$ is the column vector

$$(e \oplus r(i), f, g, h).$$

In this way, columns $W(4), \dots, W(43)$ are generated from the initial four columns.

The **round key** for the i th round consists of the columns

$$W(4i), W(4i + 1), W(4i + 2), W(4i + 3).$$

8.2.6 The Construction of the S-Box

Although the S-box is implemented as a lookup table, it has a simple mathematical description. Start with a byte $x_7x_6x_5x_4x_3x_2x_1x_0$, where each x_i is a binary bit.

Compute its inverse in $GF(2^8)$, as in [Section 3.11](#). If the byte is 0oooooooo, there is no inverse, so we use 0oooooooo in place of its inverse. The resulting byte $y_7y_6y_5y_4y_3y_2y_1y_0$ represents an eight-dimensional column vector, with the rightmost bit y_0 in the top position. Multiply by a matrix and add the column vector $(1, 1, 0, 0, 0, 1, 1, 0)$ to obtain a vector $(z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7)$ as follows:

$$\begin{array}{cccccccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & y_0 & 1 & z_0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & y_1 & 1 & z_1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & y_2 & 0 & z_2 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & y_3 & 0 & z_3 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & y_4 & 0 & z_4 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & y_5 & 1 & z_5 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & y_6 & 1 & z_6 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & y_7 & 0 & z_7 \end{array} + \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \end{array} = \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \end{array} .$$

The byte $z_7z_6z_5z_4z_3z_2z_1z_0$ is the entry in the S-box.

For example, start with the byte 11001011. Its inverse in $GF(2^8)$ is 00000100, as we calculated in [Section](#)

3.11. We now calculate

$$\begin{array}{ccccccccc}
 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0
 \end{array}
 + \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} = \begin{array}{c} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{array} .$$

This yields the byte **00011111**. The first four bits **1100** represent **12** in binary and the last four bits **1011** represent **11** in binary. Add **1** to each of these numbers (since the first row and column are numbered **0**) and look in the **13th** row and **12th** column of the S-box. The entry is **31**, which in binary is **00011111**.

Some of the considerations in the design of the S-box were the following. The map $x \mapsto x^{-1}$ was used to achieve nonlinearity. However, the simplicity of this map could possibly allow certain attacks, so it was combined with multiplication by the matrix and adding the vector, as described previously. The matrix was chosen mostly because of its simple form (note how the rows are shifts of each other). The vector was chosen so that no input ever equals its S-box output or the complement of its S-box output (complementation means changing each 1 to 0 and each 0 to 1).

8.3 Decryption

Each of the steps SubBytes, ShiftRows, MixColumns, and AddRoundKey is invertible:

1. The inverse of SubBytes is another lookup table, called **InvSubBytes**.
2. The inverse of ShiftRows is obtained by shifting the rows to the right instead of to the left, yielding **InvShiftRows**.
3. The inverse of MixColumns exists because the 4×4 matrix used in MixColumns is invertible. The transformation **InvMixColumns** is given by multiplication by the matrix

00001110	00001011	00001101	00001001
00001001	00001110	00001011	00001101
00001101	00001001	00001110	00001011
00001011	00001101	00001001	00001110

4. AddRoundKey is its own inverse.

The Rijndael encryption consists of the steps

ARK

SB, SR, MC, ARK

...

SB, SR, MC, ARK

SB, SR, ARK.

Recall that MC is missing in the last round.

To decrypt, we need to run through the inverses of these steps in the reverse order. This yields the following preliminary version of decryption:

ARK, ISR, ISB

ARK, IMC, ISR, ISB

...

ARK, IMC, ISR, ISB

ARK.

However, we want to rewrite this decryption in order to make it look more like encryption.

Observe that applying SB then SR is the same as first applying SR then SB. This happens because SB acts one byte at a time and SR permutes the bytes.

Correspondingly, the order of ISR and ISB can be reversed.

We also want to reverse the order of ARK and IMC, but this is not possible. Instead, we proceed as follows.

Applying MC and then ARK to a matrix $(c_{i,j})$ is given as

$$(c_{i,j}) \rightarrow (m_{i,j})(c_{i,j}) \rightarrow (e_{i,j}) = (m_{i,j})(c_{i,j}) \oplus (k_{i,j}),$$

where $(m_{i,j})$ is a the 4×4 matrix in MixColumns and $(k_{i,j})$ is the round key matrix. The inverse is obtained by solving $(e_{i,j}) = (m_{i,j})(c_{i,j}) \oplus (k_{i,j})$ for $(c_{i,j})$ in terms of $(e_{i,j})$, namely,

$(c_{i,j}) = (m_{i,j})^{-1}(e_{i,j}) \oplus (m_{i,j})^{-1}(k_{i,j})$. Therefore, the process is

$$(e_{i,j}) \rightarrow (m_{i,j})^{-1}(e_{i,j}) \rightarrow (m_{i,j})^{-1}(e_{i,j}) \oplus (k'_{i,j}),$$

where $(k'_{i,j}) = (m_{i,j})^{-1}(k_{i,j})$. The first arrow is simply InvMixColumns applied to $(e_{i,j})$. If we let **InvAddRoundKey** be XORing with $(k'_{i,j})$, then we have that the inverse of “MC then ARK” is “ IMC then IARK.” Therefore, we can replace the steps “ARK then IMC” with the steps “IMC then IARK” in the preceding decryption sequence.

We now see that decryption is given by

ARK, ISB, ISR

IMC, IARK, ISB, ISR

...

IMC, IARK, ISB, ISR

ARK.

Regroup the lines to obtain the final version:

Rijndael Decryption

1. ARK, using the 10th round key
2. Nine rounds of ISB, ISR, IMC, IARK, using round keys 9 to 1
3. A final round: ISB, ISR, ARK, using the 0th round key

8.3-3 Full Alternative Text

Therefore, the decryption is given by essentially the same structure as encryption, but SubBytes, ShiftRows, and MixColumns are replaced by their inverses, and AddRoundKey is replaced by InvAddRoundKey, except in the initial and final steps. Of course, the round keys are used in the reverse order, so the first ARK uses the 10th round key, and the last ARK uses the 0th round key.

The preceding shows why the MixColumns is omitted in the last round. Suppose it had been left in. Then the encryption would start ARK, SB, SR, MC, ARK, ..., and it would end with ARK, SB, SR, MC, ARK. Therefore, the beginning of the decryption would be (after the reorderings) IMC, IARK, ISB, ISR, This means the decryption would have an unnecessary IMC at the beginning, and this would have the effect of slowing down the algorithm.

Another way to look at encryption is that there is an initial ARK, then a sequence of alternating half rounds

(SB, SR), (MC, ARK), (SB, SR), . . . , (MC, ARK), (SB, SR),

followed by a final ARK. The decryption is ARK, followed by a sequence of alternating half rounds

(ISB, ISR), (IMC, IARK), (ISB, ISR), . . . , (IMC, IARK), (ISB, ISR),

followed by a final ARK. From this point of view, we see that a final MC would not fit naturally into any of the half rounds, and it is natural to leave it out.

On eight-bit processors, decryption is not quite as fast as encryption. This is because the entries in the 4×4 matrix for InvMixColumns are more complex than those for MixColumns, and this is enough to make decryption take around 30% longer than encryption for these processors. However, in many applications, decryption is not needed, for example, when CFB mode (see [Section 6.3](#)) is used. Therefore, this is not considered to be a significant drawback.

The fact that encryption and decryption are not identical processes leads to the expectation that there are no weak keys, in contrast to DES (see [Exercise 5 in Chapter 7](#)) and several other algorithms.

8.4 Design Considerations

The Rijndael algorithm is not a Feistel system (see Sections 7.1 and 7.2). In a Feistel system, half the bits are moved but not changed during each round. In Rijndael, all bits are treated uniformly. This has the effect of diffusing the input bits faster. It can be shown that two rounds are sufficient to obtain full diffusion, namely, each of the 128 output bits depends on each of the 128 input bits.

The S-box was constructed in an explicit and simple algebraic way so as to avoid any suspicions of trapdoors built into the algorithm. The desire was to avoid the mysteries about the S-boxes that haunted DES. The Rijndael S-box is highly nonlinear, since it is based on the mapping $x \mapsto x^{-1}$ in $GF(2^8)$. It is excellent at resisting differential and linear cryptanalysis, as well as more recently studied methods called interpolation attacks.

The ShiftRows step was added to resist two recently developed attacks, namely truncated differentials and the Square attack (Square was a predecessor of Rijndael).

The MixColumns causes diffusion among the bytes. A change in one input byte in this step always results in all four output bytes changing. If two input bytes are changed, at least three output bytes are changed.

The Key Schedule involves nonlinear mixing of the key bits, since it uses the S-box. The mixing is designed to resist attacks where the cryptanalyst knows part of the key and tries to deduce the remaining bits. Also, it aims to ensure that two distinct keys do not have a large number of round keys in common. The round constants

are used to eliminate symmetries in the encryption process by making each round different.

The number of rounds was chosen to be 10 because there are attacks that are better than brute force up to six rounds. No known attack beats brute force for seven or more rounds. It was felt that four extra rounds provide a large enough margin of safety. Of course, the number of rounds could easily be increased if needed.

8.5 Exercises

1. Suppose the key for round 0 in AES consists of 128 bits, each of which is 0.

1. Show that the key for the first round is $W(4), W(5), W(6), W(7)$, where

$$W(4) = W(5) = W(6) = W(7) = \begin{array}{c} 01100010 \\ 01100011 \\ 01100011 \\ 01100011 \end{array}.$$

2. Show that $W(8) = W(10) \neq W(9) = W(11)$ (Hint: This can be done without computing $W(8)$ explicitly).

2. Suppose the key for round 0 in AES consists of 128 bits, each of which is 1.

1. Show that the key for the first round is $W(4), W(5), W(6), W(7)$, where

$$W(5) = W(7) = \begin{array}{c} 00010111 \\ 00010110 \\ 00010110 \\ 00010110 \end{array},$$
$$W(4) = W(6) = \begin{array}{c} 11101000 \\ 11101001 \\ 11101001 \\ 11101001 \end{array}.$$

Note that $W(5) = W(4)$ = the complement of $W(5)$ (the complement can be obtained by XOR ing with a string of all 1s).

2. Show that $W(10) = W(8)$ and that $W(11) = W(9)$ (Hints: $W(5) \oplus W(6)$ is a string of all 1s. Also, the relation $A \oplus B = A \oplus B$ might be useful.)

3. Let $f(x)$ be a function from binary strings (of a fixed length N) to binary strings. For the purposes of this problem, let's say that $f(x)$ has the *equal difference property* if the following is satisfied: Whenever x_1, x_2, x_3, x_4 are binary strings of length N that satisfy $x_1 \oplus x_2 = x_3 \oplus x_4$, then

$$f(x_1) \oplus f(x_2) = f(x_3) \oplus f(x_4).$$

1. Show that if $\alpha, \beta \in GF(2^8)$ and $f(x) = \alpha x + \beta$ for all $x \in GF(2^8)$, then $f(x)$ has the equal difference property.
2. Show that the ShiftRows Transformation, the MixColumns Transformation, and the RoundKey Addition have the equal difference property.
4.
 1. Suppose we remove all SubBytes Transformation steps from the AES algorithm. Show that the resulting AES encryption would then have the equal difference property defined in [Exercise 3](#).
 2. Suppose we are in the situation of part (a), with all SubBytes Transformation steps removed. Let x_1 and x_2 be two 128-bit plaintext blocks and let $E(x_1)$ and $E(x_2)$ be their encryptions under this modified AES scheme. Show that $E(x_1) \oplus E(x_2)$ equals the result of encrypting $x_1 \oplus x_2$ using only the ShiftRows and MixColumns Transformations (that is, both the RoundKey Addition and the SubBytes Transformation are missing). In particular, $E(x_1) \oplus E(x_2)$ is independent of the key.
 3. Suppose we are in the situation of part (a), and Eve knows x_1 and $E(x_1)$ for some 128-bit string x . Describe how she can decrypt any message $E(x_2)$ (your solution should be much faster than using brute force or making a list of all encryptions). (Remark: This shows that the SubBytes transformation is needed to prevent the equal difference property. See also [Exercise 5](#).)
5. Let $x_1 = 00000000$, $x_2 = 00000001$, $x_3 = 00000010$, $x_4 = 00000011$. Let $SB(x)$ denote the SubBytes Transformation of x . Show that

$$SB(x_1) \oplus SB(x_2) = 00011111 \neq 00001100 = SB(x_3) \oplus SB(x_4).$$

Conclude that the SubBytes Transformation is not an affine map (that is, a map of the form $\alpha x + \beta$) from $GF(2^8)$ to $GF(2^8)$.
 (Hint: See [Exercise 3\(a\)](#).)

6. Your friend builds a very powerful computer that uses brute force to find a 56-bit DES key in 1 hour, so you make an even better machine that can try 2^{56} AES keys in 1 second. How long will this machine take to try all 2^{128} AES keys?

Chapter 9 The RSA Algorithm

9.1 The RSA Algorithm

Alice wants to send a message to Bob, but they have not had previous contact and they do not want to take the time to send a courier with a key. Therefore, all information that Alice sends to Bob will potentially be obtained by the evil observer Eve. However, it is still possible for a message to be sent in such a way that Bob can read it but Eve cannot.

With all the previously discussed methods, this would be impossible. Alice would have to send a key, which Eve would intercept. She could then decrypt all subsequent messages. The possibility of the present scheme, called a **public key cryptosystem**, was first publicly suggested by Diffie and Hellman in their classic 1976 paper [Diffie-Hellman]. However, they did not yet have a practical implementation (although they did present an alternative key exchange procedure that works over public channels; see [Section 10.4](#)). In the next few years, several methods were proposed. The most successful, based on the idea that factorization of integers into their prime factors is hard, was proposed by Rivest, Shamir, and Adleman in 1977 and is known as the RSA algorithm.

It had long been claimed that government cryptographic agencies had discovered the RSA algorithm several years earlier, but secrecy rules prevented them from releasing any evidence. Finally, in 1997, documents released by CESG, a British cryptographic agency, showed that in 1970, James Ellis had discovered public key cryptography, and in 1973, Clifford Cocks had written an

internal document describing a version of the RSA algorithm in which the encryption exponent e (see the discussion that follows) was the same as the modulus n .

Here is how the RSA algorithm works. Bob chooses two distinct large primes p and q and multiplies them together to form

$$n = pq.$$

He also chooses an encryption exponent e such that

$$\gcd(e, (p-1)(q-1)) = 1.$$

He sends the pair (n, e) to Alice but keeps the values of p and q secret. In particular, Alice, who could possibly be an enemy of Bob, never needs to know p and q to send her message to Bob securely. Alice writes her message as a number m . If m is larger than n , she breaks the message into blocks, each of which is less than n . However, for simplicity, let's assume for the moment that $m < n$. Alice computes

$$c \equiv m^e \pmod{n}$$

and sends c to Bob. Since Bob knows p and q , he can compute $(p-1)(q-1)$ and therefore can find the decryption exponent d with

$$de \equiv 1 \pmod{(p-1)(q-1)}.$$

As we'll see later,

$$m \equiv c^d \pmod{n},$$

so Bob can read the message.

We summarize the algorithm in the following table.

The RSA Algorithm

1. Bob chooses secret primes p and q and computes $n = pq$.
2. Bob chooses e with $\gcd(e, (p-1)(q-1)) = 1$.
3. Bob computes d with $de \equiv 1 \pmod{(p-1)(q-1)}$.
4. Bob makes n and e public, and keeps p, q, d secret.
5. Alice encrypts m as $c \equiv m^e \pmod{n}$ and sends c to Bob.
6. Bob decrypts by computing $m \equiv c^d \pmod{n}$.

9.1-1 Full Alternative Text

Example

Bob chooses

$$p = 885320963, \quad q = 238855417.$$

Then

$$n = p \cdot q = 211463707796206571.$$

Let the encryption exponent be

$$e = 9007.$$

The values of n and e are sent to Alice.

Alice's message is *cat*. We will depart from our earlier practice of numbering the letters starting with $a = 0$; instead, we start the numbering at $a = 01$ and continue through $z = 26$. We do this because, in the previous method, if the letter a appeared at the beginning of a message, it would yield a message number m starting with 00, so the a would disappear.

The message is therefore

$$m = 30120.$$

Alice computes

$$c \equiv m^e \equiv 30120^{9007} \equiv 113535859035722866 \pmod{n}.$$

She sends c to Bob.

Since Bob knows p and q , he knows $(p - 1)(q - 1)$. He uses the extended Euclidean algorithm (see [Section 3.2](#)) to compute d such that

$$de \equiv 1 \pmod{(p - 1)(q - 1)}.$$

The answer is

$$d = 116402471153538991.$$

Bob computes

$$c^d \equiv 113535859035722866^{116402471153538991} \equiv 30120 \pmod{n},$$

so he obtains the original message.

For more examples, see Examples 24–30 in the Computer Appendices.

There are several aspects that need to be explained, but perhaps the most important is why $m \equiv c^d \pmod{n}$. Recall Euler's theorem ([Section 3.6](#)): If $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$. In our case, $\phi(n) = \phi(pq) = (p - 1)(q - 1)$. Suppose $\gcd(m, n) = 1$. This is very likely the case; since p and q are large, m probably has neither as a factor. Since $de \equiv 1 \pmod{\phi(n)}$, we can write $de = 1 + k\phi(n)$ for some integer k . Therefore,

$$c^d \equiv (m^e)^d \equiv m^{1+k\phi(n)} \equiv m \cdot (m^{\phi(n)})^k \equiv m \cdot 1^k \equiv m \pmod{n}.$$

We have shown that Bob can recover the message. If $\gcd(m, n) \neq 1$, Bob still recovers the message. See [Exercise 37](#).

What does Eve do? She intercepts n, e, c . She does not know p, q, d . We assume that Eve has no way of

factoring n . The obvious way of computing d requires knowing $\phi(n)$. We show later that this is equivalent to knowing p and q . Is there another way? We will show that if Eve can find d , then she can probably factor n . Therefore, it is unlikely that Eve finds the decryption exponent d .

Since Eve knows $c \equiv m^e \pmod{n}$, why doesn't she simply take the e th root of c ? This works well if we are not working mod n but is very difficult in our case. For example, if you know that $m^3 \equiv 3 \pmod{85}$, you cannot calculate the cube root of 3, namely $1.4422\dots$, on your calculator and then reduce mod 85. Of course, a case-by-case search would eventually yield $m = 7$, but this method is not feasible for large n .

How does Bob choose p and q ? They should be chosen at random, independently of each other. How large depends on the level of security needed, but it seems that they should have at least 300 digits. For reasons that we discuss later, it is perhaps best if they are of slightly different lengths. When we discuss primality testing, we'll see that finding such primes can be done fairly quickly (see also [Section 3.6](#)). A few other tests should be done on p and q to make sure they are not bad. For example, if $p - 1$ has only small prime factors, then n is easy to factor by the $p - 1$ method (see [Section 9.3](#)), so p should be rejected and replaced with another prime.

Why does Bob require $\gcd(e, (p-1)(q-1)) = 1$? Recall (see [Section 3.3](#)) that $de \equiv 1 \pmod{(p-1)(q-1)}$ has a solution d if and only if $\gcd(e, (p-1)(q-1)) = 1$. Therefore, this condition is needed in order for d to exist. The extended Euclidean algorithm can be used to compute d quickly. Since $p - 1$ is even, $e = 2$ cannot be used; one might be tempted to use $e = 3$. However, there are dangers in using small values of e (see [Section 9.2](#), Computer Problem 14, and [Section 23.3](#)), so something larger is

usually recommended. Also, if e is a moderately large prime, then there is no difficulty ensuring that $\gcd(e, (p-1)(q-1)) = 1$. It is now generally recommended that $e \geq 65537$. Among the data collected for [Lenstra2012 et al.] is the distribution of RSA encryption exponents that is given in Table 9.1.

Table 9.1 RSA Encryption Exponents

e	Percentage
65537	95.4933%
17	3.1035%
41	0.4574%
3	0.3578%
19	0.1506%
25	0.1339%
5	0.1111%
7	0.0596%
11	0.0313%
257	0.0241%
other	0.0774%

Table 9.1 Full Alternative Text

In the encryption process, Alice calculates $m^e \pmod{n}$. Recall that this can be done fairly quickly and without large memory, for example, by successive squaring. See Section 3.5. This is definitely an advantage of modular arithmetic: If Alice tried to calculate m^e first, then

reduce mod n , it is possible that recording m^e would overflow her computer's memory. Similarly, the decryption process of calculating $c^d \pmod{n}$ can be done efficiently. Therefore, all the operations needed for encryption and decryption can be done quickly (i.e., in time a power of $\log n$). The security is provided by the assumption that n cannot be factored.

We made two claims. We justify them here. Recall that the point of these two claims was that finding $\phi(n)$ or finding the decryption exponent d is essentially as hard as factoring n . Therefore, if factoring is hard, then there should be no fast, clever way of finding d .

Claim 1: Suppose $n = pq$ is the product of two distinct primes. If we know n and $\phi(n)$, then we can quickly find p and q .

Note that

$$n - \phi(n) + 1 = pq - (p - 1)(q - 1) + 1 = p + q.$$

Therefore, we know pq and $p + q$. The roots of the polynomial

$$X^2 - (n - \phi(n) + 1)X + n = X^2 - (p + q)X + pq = (X - p)(X - q)$$

are p and q , but they can also be calculated by the quadratic formula:

$$p, q = \frac{(n - \phi(n) + 1) \pm \sqrt{(n - \phi(n) + 1)^2 - 4n}}{2}.$$

This yields p and q .

For example, suppose $n = 221$ and we know that $\phi(n) = 192$. Consider the quadratic equation

$$X^2 - 30X + 221.$$

The roots are

$$p, q = \frac{30 \pm \sqrt{30^2 - 4 \cdot 221}}{2} = 13, 17.$$

For another example, see Example 31 in the Computer Appendices.

Claim 2: If we know d and e , then we can probably factor n .

In the discussion of factorization methods in [Section 9.4](#), we show that if we have an exponent $r > 0$ such that $a^r \equiv 1 \pmod{n}$ for several a with $\gcd(a, n) = 1$, then we can probably factor n . Since $de - 1$ is a multiple of $\phi(n)$, say $de - 1 = k\phi(n)$, we have

$$a^{de-1} \equiv (a^{\phi(n)})^k \equiv 1 \pmod{n}$$

whenever $\gcd(a, n) = 1$. The $a^r \equiv 1$ factorization method can now be applied.

For an example, see Example 32 in the Computer Appendices.

Claim 2': If e is small or medium-sized (for example, if e has several fewer digits than \sqrt{n}) and we know d , then we can factor n .

We use the following procedure:

1. Compute $k = \lceil (de - 1)/n \rceil$ (that is, round up to the nearest integer).
2. Compute

$$\phi(n) = \frac{de - 1}{k}.$$

3. Solve the quadratic equation

$$X^2 - (n + 1 - \phi(n))X + n = 0.$$

The solutions are p and q .

Example

Let $n = 670726081$ and $e = 257$. We discover that $d = 524523509$. Compute

$$\frac{de - 1}{n} = 200.98\ldots$$

Therefore, $k = 201$. Then

$$\phi(n) = \frac{de - 1}{201} = 670659412.$$

The roots of the equation

$$X^2 - (n + 1 - \phi(n))X + n = X^2 - 66670X + 670726081 = 0$$

are $12347.00\ldots$ and $54323.00\ldots$, and we can check that $n = 12347 \times 54323$.

Why does this work? We know that

$de \equiv 1 \pmod{(p-1)(q-1)}$, so we write

$de = 1 + (p-1)(q-1)k$. Since

$d < (p-1)(q-1)$, we know that

$$(p-1)(q-1)k < de < (p-1)(q-1)e,$$

so $k < e$. We have

$$\begin{aligned} k &= \frac{de - 1}{(p-1)(q-1)} > \frac{de - 1}{n} = \frac{(p-1)(q-1)k}{n} = \frac{(pq - p - q + 1)k}{n} \\ &= k - \frac{(p+q-1)k}{n}. \end{aligned}$$

Usually, both p and q are approximately \sqrt{n} . In practice, e and therefore k (which is less than e) are much smaller than \sqrt{n} . Therefore, $(p+q-1)k$ is much smaller than n , which means that $(p+q-1)k/n < 1$ and it rounds off to 0.

Once we have k , we use $de - 1 = \phi(n)k$ to solve for $\phi(n)$. As we have already seen, once we know n and $\phi(n)$, we can find p and q by solving the quadratic equation.

One way the RSA algorithm can be used is when there are several banks, for example, that want to be able to

send financial data to each other. If there are several thousand banks, then it is impractical for each pair of banks to have a key for secret communication. A better way is the following. Each bank chooses integers n and e as before. These are then published in a public book. Suppose bank A wants to send data to bank B. Then A looks up B's n and e and uses them to send the message. In practice, the RSA algorithm is not quite fast enough for sending massive amounts of data. Therefore, the RSA algorithm is often used to send a key for a faster encryption method such as AES.

PGP (= Pretty Good Privacy) used to be a standard method for encrypting email. When Alice sends an email message to Bob, she first signs the message using a digital signature algorithm such as those discussed in [Chapter 13](#). She then encrypts the message using a block cipher such as triple DES or AES (other choices are IDEA or CAST-128) with a randomly chosen 128-bit key (a new random key is chosen for each transmission). She then encrypts this key using Bob's public RSA key (other public key methods can also be used). When Bob receives the email, he uses his private RSA exponent to decrypt the random key. Then he uses this random key to decrypt the message, and he checks the signature to verify that the message is from Alice. For more discussion of PGP, see [Section 15.6](#).

9.2 Attacks on RSA

In practice, the RSA algorithm has proven to be effective, as long as it is implemented correctly. We give a few possible implementation mistakes in the Exercises. Here are a few other potential difficulties. For more about attacks on RSA, see [Boneh].

Theorem

Let $n = pq$ have m digits. If we know the first $m/4$, or the last $m/4$, digits of p , we can efficiently factor n .

In other words, if p and q have 300 digits, and we know the first 150 digits, or the last 150 digits, of p , then we can factor n . Therefore, if we choose a random starting point to choose our prime p , the method should be such that a large amount of p is not predictable. For example, suppose we take a random 150-digit number N and test numbers of the form $N \cdot 10^{150} + k$, $k = 1, 3, 5, \dots$, for primality until we find a prime p (which should happen for $k < 10000$). An attacker who knows that this method is used will know 147 of the last 150 digits (they will all be 0 except for the last three or four digits). Trying the method of the theorem for the various values of $k < 10000$ will eventually lead to the factorization of n .

For details of the preceding result, see [Coppersmith2]. A related result is the following.

Theorem

Suppose (n, e) is an RSA public key and n has m digits. Let d be the decryption exponent. If we have at least the last $m/4$ digits of d , we can efficiently find d in time that is linear in $e \log_2 e$.

This means that the time to find d is bounded as a function linear in $e \log_2 e$. If e is small, it is therefore quite fast to find d when we know a large part of d . If e is large, perhaps around n , the theorem is no better than a case-by-case search for d . For details, see [Boneh et al.].

9.2.1 Low Exponent Attacks

Low encryption or decryption exponents are tempting because they speed up encryption or decryption. However, there are certain dangers that must be avoided. One pitfall of using $e = 3$ is given in Computer Problem 14. Another difficulty is discussed in [Chapter 23](#) (Lattice Methods). These problems can be avoided by using a somewhat higher exponent. One popular choice is $e = 65537 = 2^{16} + 1$. This is prime, so it is likely that it is relatively prime to $(p - 1)(q - 1)$. Since it is one more than a power of 2, exponentiation to this power can be done quickly: To calculate x^{65537} , square x sixteen times, then multiply the result by x .

The decryption exponent d should of course be chosen large enough that brute force will not find it. However, even more care is needed, as the following result shows. One way to obtain desired properties of d is to choose d first, then find e with $de \equiv 1 \pmod{\phi(n)}$.

Suppose Bob wants to be able to decrypt messages quickly, so he chooses a small value of d . The following theorem of M. Wiener [Wiener] shows that often Eve can then find d easily. In practice, if the inequalities in the hypotheses of the proposition are weakened, then Eve can still use the method to obtain d in many cases.

Therefore, it is recommended that d be chosen fairly large.

Theorem

Suppose p, q are primes with $q < p < 2q$. Let $n = pq$ and let $1 \leq d, e < \phi(n)$ satisfy

$de \equiv 1 \pmod{(p-1)(q-1)}$. If $d < \frac{1}{3}n^{1/4}$, then d can be calculated quickly (that is, in time polynomial in $\log n$).

Proof. Since $q^2 < pq = n$, we have $q < \sqrt{n}$. Therefore, since $p < 2q$,

$$n - \phi(n) = pq - (p-1)(q-1) = p + q - 1 < 3q < 3\sqrt{n}.$$

Write $ed = 1 + \phi(n)k$ for some integer $k \geq 1$. Since $e < \phi(n)$, we have

$$\phi(n)k < ed < \frac{1}{3}\phi(n)n^{1/4},$$

so $k < \frac{1}{3}n^{1/4}$. Therefore,

$$kn - ed = k(n - \phi(n)) - 1 < k(n - \phi(n)) < \frac{1}{3}n^{1/4}(3\sqrt{n}) = n^{3/4}.$$

Also, since $k(n - \phi(n)) - 1 > 0$, we have
 $kn - ed > 0$. Dividing by dn yields

$$0 < \frac{k}{d} - \frac{e}{n} < \frac{1}{dn^{1/4}} < \frac{1}{3d^2},$$

since $3d < n^{1/4}$ by assumption.

We now need a result about continued fractions. Recall from [Section 3.12](#) that if x is a positive real number and k and d are positive integers with

$$\left| \frac{k}{d} - x \right| < \frac{1}{2d^2},$$

then k/d arises from the continued fraction expansion of x . Therefore, in our case, k/d arises from the continued fraction expansion of e/n . Therefore, Eve does the following:

1. Computes the continued fraction of e/n . After each step, she obtains a fraction A/B .
2. Eve uses $k = A$ and $d = B$ to compute $C = (ed - 1)/k$. (Since $ed = 1 + \phi(n)k$, this value of C is a candidate for $\phi(n)$.)
3. If C is not an integer, she proceeds to the next step of the continued fraction.
4. If C is an integer, then she finds the roots r_1, r_2 of $X^2 - (n - C + 1)X + n$. (Note that this is possibly the equation $X^2 - (n - \phi(n) + 1)X + n = (X - p)(X - q)$ from earlier.) If r_1 and r_2 are integers, then Eve has factored n . If not, then Eve proceeds to the next step of the continued fraction algorithm.

Since the number of steps in the continued fraction expansion of e/n is at most a constant times $\log n$, and since the continued fraction algorithm stops when the fraction e/n is reached, the algorithm terminates quickly. Therefore, Eve finds the factorization of n quickly.

Remarks

Recall that the rational approximations to a number x arising from the continued fraction algorithm are alternately larger than x and smaller than x . Since $0 < \frac{k}{d} - \frac{e}{n}$, we only need to consider every second fraction arising from the continued fraction.

What happens if Eve reaches e/n without finding the factorization of n ? This means that the hypotheses of the proposition are not satisfied. However, it is possible that sometimes the method will yield the factorization of n even when the hypotheses fail.

Example

Let $n = 1966981193543797$ and
 $e = 323815174542919$. The continued fraction of e/n
is

$$\begin{aligned} & [0; 6, 13, 2, 3, 1, 3, 1, 9, 1, 36, 5, 2, 1, 6, 1, 43, 13, 1, 10, 11, 2, 1, 9, 5] \\ &= \cfrac{1}{6 + \cfrac{1}{13 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{1 + \dots}}}}}}}. \end{aligned}$$

The first fraction is $1/6$, so we try $k = 1, d = 6$. Since d must be odd, we discard this possibility.

By the remark, we may jump to the third fraction:

$$\cfrac{1}{6 + \cfrac{1}{13 + \cfrac{1}{2}}} = \cfrac{27}{164}.$$

Again, we discard this since d must be odd.

The fifth fraction is $121/735$. This gives
 $C = (e \cdot 735 - 1)/121$, which is not an integer.

The seventh fraction is $578/3511$. This gives
 $C = 1966981103495136$ as the candidate for $\phi(n)$.
The roots of

$$X^2 - (n - C + 1)X + n$$

are 37264873 and 52783789 , to several decimal places
of accuracy. Since

$$n = 37264873 \times 52783789,$$

we have factored n .

9.2.2 Short Plaintext

A common use of RSA is to transmit keys for use in DES, AES, or other symmetric cryptosystems. However, a naive implementation could lead to a loss of security.

Suppose a 56-bit DES key is written as a number $m \approx 10^{17}$. This is encrypted with RSA to obtain $c \equiv m^e \pmod{n}$. Although m is small, the ciphertext c is probably a number of the same size as n , so perhaps around 200 digits. However, Eve attacks the system as follows. She makes two lists:

$$1. cx^{-e} \pmod{n} \text{ for all } x \text{ with } 1 \leq x \leq 10^9.$$

$$2. y^e \pmod{n} \text{ for all } y \text{ with } 1 \leq y \leq 10^9.$$

She looks for a match between an element on the first list and an element on the second list. If she finds one, then she has $cx^{-e} \equiv y^e$ for some x, y . This yields

$$c \equiv (xy)^e \pmod{n},$$

so $m \equiv xy \pmod{n}$. Is this attack likely to succeed?

Suppose m is the product of two integers x and y , both less than 10^9 . Then these x, y will yield a match for Eve. Not every m will have this property, but many values of m are the product of two integers less than 10^9 . For these, Eve will obtain m .

This attack is much more efficient than trying all 10^{17} possibilities for m , which is nearly impossible on one computer, and would take a long time even with several computers working in parallel. In the present attack, Eve needs to compute and store a list of length 10^9 , then compute the elements on the other list and check each one against the first list. Therefore, Eve performs approximately 2×10^9 computations (and compares with the list up to 10^9 times). This is easily possible on a single computer. For more on this attack, see [Boneh-Joux-Nguyen].

It is easy to prevent this attack. Instead of using a small value of m , adjoin some random digits to the beginning and end of m so as to form a much longer plaintext.

When Bob decrypts the ciphertext, he simply removes these random digits and obtains m .

A more sophisticated method of preprocessing the plaintext, namely Optimal Asymmetric Encryption Padding (OAEP), was introduced by Bellare and Rogaway [Bellare-Rogaway2] in 1994. Suppose Alice wants to send a message m to Bob, whose RSA public key is (n, e) , where n has k bits. Two positive integers k_0 and k_1 are specified in advance, with $k_0 + k_1 < k$. Alice's message is allowed to have $k - k_0 - k_1$ bits. Typical values are $k = 1024$, $k_0 = k_1 = 128$, $k - k_0 - k_1 = 768$. Let G be a function that inputs strings of k_0 bits and outputs strings of $k - k_0$ bits. Let H be a function that inputs $k - k_0$ bits and outputs k_0 bits. The functions G and H are usually constructed from hash functions (see Chapter 11 for a discussion of hash functions). To encrypt m , Alice first expands it to length $k - k_0$ by adjoining k_1 zero bits. The result is denoted $m0^{k_1}$. She then chooses a random string r of k_0 bits and computes

$$x_1 = m0^{k_1} \oplus G(r), \quad x_2 = r \oplus H(x_1).$$

If the concatenation $x_1 \parallel x_2$ is a binary number larger than n , Alice chooses a new random number r and computes new values for x_1 and x_2 . As soon as she obtains $x_1 \parallel x_2 < n$ (this has a probability of at least $1/2$ of happening for each r , as long as $G(r)$ produces fairly random outputs), she forms the ciphertext

$$E(m) = (x_1 \parallel x_2)^e \pmod{n}.$$

To decrypt a ciphertext c , Bob uses his private RSA decryption exponent d to compute $c^d \pmod{n}$. The result is written in the form

$$c^d \pmod{n} = y_1 \parallel y_2,$$

where y_1 has $k - k_0$ bits and y_2 has k_0 bits. Bob then computes

$$m0^{k_1} = y_1 \oplus G(H(y_1) \oplus y_2).$$

The correctness of this decryption can be justified as follows. If the ciphertext is the encryption of m , then

$$y_1 = x_1 = m0^{k_1} \oplus G(r) \quad \text{and} \quad y_2 = x_2 = r \oplus H(x_1).$$

Therefore,

$$H(y_1) \oplus y_2 = H(x_1) \oplus r \oplus H(x_1) = r$$

and

$$y_1 \oplus G(H(y_1) \oplus y_2) = x_1 \oplus G(r) = m0^{k_1}.$$

Bob removes the k_1 zero bits from the end of $m0^{k_1}$ and obtains m . Also, Bob has check on the integrity of the ciphertext. If there are not k_1 zeros at the end, then the ciphertext does not correspond to a valid encryption.

This method is sometimes called plaintext-aware encryption. Note that the padding with x_2 depends on the message m and on the random parameter r . This makes chosen ciphertext attacks on the system more difficult. It also is used for ciphertext indistinguishability. See [Section 4.5](#).

For discussion of the security of OAEP, see [Shoup].

9.2.3 Timing Attacks

Another type of attack on RSA and similar systems was discovered by Paul Kocher in 1995, while he was an undergraduate at Stanford. He showed that it is possible to discover the decryption exponent by carefully timing the computation times for a series of decryptions. Though there are ways to thwart the attack, this development was unsettling. There had been a general

feeling of security since the mathematics was well understood. Kocher's attack demonstrated that a system could still have unexpected weaknesses.

Here is how the timing attack works. Suppose Eve is able to observe Bob decrypt several ciphertexts y . She times how long this takes for each y . Knowing each y and the time required for it to be decrypted will allow her to find the decryption exponent d . But first, how could Eve obtain such information? There are several situations where encrypted messages are sent to Bob and his computer automatically decrypts and responds. Measuring the response times suffices for the present purposes.

We need to assume that we know the hardware being used to calculate $y^d \pmod{n}$. We can use this information to calculate the computation times for various steps that potentially occur in the process.

Let's assume that $y^d \pmod{n}$ is computed by an algorithm given in [Exercise 56 in Chapter 3](#), which is as follows:

Let $d = b_1 b_2 \dots b_w$ be written in binary (for example, when $x = 1011$, we have $b_1 = 1, b_2 = 0, b_3 = 1, b_4 = 1$). Let y and n be integers. Perform the following procedure:

1. Start with $k = 1$ and $s_1 = 1$.
2. If $b_k = 1$, let $r_k \equiv s_k y \pmod{n}$. If $b_k = 0$, let $r_k = s_k$.
3. Let $s_{k+1} \equiv r_k^2 \pmod{n}$.
4. If $k = w$, stop. If $k < w$, add 1 to k and go to (2).

Then $r_w \equiv y^d \pmod{n}$.

Note that the multiplication $s_k y$ occurs only when the bit $b_k = 1$. In many situations, there is a reasonably large

variation in how long this multiplication takes. We assume this is the case here.

Before we continue, we need a few facts from probability. Suppose we have a random process that produces real numbers t as outputs. For us, t will be the time it takes for the computer to complete a calculation, given a random input y . The mean is the average value of these outputs. If we record outputs t_1, \dots, t_n , the mean should be approximately $m = (t_1 + \dots + t_n)/n$. The variance for the random process is approximated by

$$\text{Var}(\{t_i\}) = \frac{(t_1 - m)^2 + \dots + (t_n - m)^2}{n}.$$

The standard deviation is the square root of the variance and gives a measure of how much variation there is in the values of the t_i 's.

The important fact we need is that when two random processes are independent, the variance for the sum of their outputs is the sum of the variances of the two processes. For example, we will break the computation done by the computer into two independent processes, which will take times t' and t'' . The total time t will be $t' + t''$. Therefore, $\text{Var}(\{t_i\})$ should be approximately $\text{Var}(\{t'_i\}) + \text{Var}(\{t''_i\})$.

Now assume Eve knows ciphertexts y_1, \dots, y_n and the times that it took to compute each $y_i^d \pmod{n}$. Suppose she knows bits b_1, \dots, b_{k-1} of the exponent d . Since she knows the hardware being used, she knows how much time was used in calculating r_1, \dots, r_{k-1} in the preceding algorithm. Therefore, she knows, for each y_i , the time t_i that it takes to compute r_k, \dots, r_w .

Eve wants to determine b_k . If $b_k = 1$, a multiplication $s_k y \pmod{n}$ will take place for each ciphertext y_i that is processed. If $b_k = 0$, there is no such multiplication.

Let t'_i be the amount of time it takes the computer to perform the multiplication $s_k y \pmod n$, though Eve does not yet know whether this multiplication actually occurs. Let $t''_i = t_i - t'_i$. Eve computes $\text{Var}(\{t_i\})$ and $\text{Var}(\{t''_i\})$. If $\text{Var}(\{t_i\}) > \text{Var}(\{t''_i\})$, then Eve concludes that $b_k = 1$. If not, $b_k = 0$. After determining b_k , she proceeds in the same manner to find all the bits.

Why does this work? If the multiplication occurs, t''_i is the amount of time it takes the computer to complete the calculation after the multiplication. It is reasonable to assume t'_i and t''_i are outputs that are independent of each other. Therefore,

$$\text{Var}(\{t_i\}) \approx \text{Var}(\{t'_i\}) + \text{Var}(\{t''_i\}) > \text{Var}(\{t''_i\}).$$

If the multiplication does not occur, t'_i is the amount of time for an operation unrelated to the computation, so it is reasonable to assume t_i and t'_i are independent. Therefore,

$$\text{Var}(\{t''_i\}) \approx \text{Var}(\{t_i\}) + \text{Var}(\{-t'_i\}) > \text{Var}(\{t_i\}).$$

Note that we couldn't use the mean in place of the variance, since the mean of $\{-t_i\}$ would be negative, so the last inequality would not hold. All that can be deduced from the mean is the total number of nonzero bits in the binary expansion of d .

The preceding gives a fairly simple version of the method. In practice, various modifications would be needed, depending on the specific situation. But the general strategy remains the same. For more details, see [Kocher]. For more on timing attacks, see [Crosby et al.].

A similar attack on RSA works by measuring the power consumed during the computations. See [Kocher et al.]. Another method, called acoustic cryptanalysis, obtains information from the high-pitched noises emitted by the electronic components of a computer during its computations. See [Genkin et al.]. Attacks such as these

and the timing attack can be prevented by appropriate design features in the physical implementation.

Timing attacks, power analysis, and acoustic cryptanalysis are examples of **side-channel attacks**, where the attack is on the implementation rather than on the basic cryptographic algorithm.

9.3 Primality Testing

Suppose we have an integer of 300 digits that we want to test for primality. We know by [Exercise 7 in Chapter 3](#) that one way is to divide by all the primes up to its square root. What happens if we try this? There are around 3×10^{147} primes less than 10^{150} . This is significantly more than the number of particles in the universe. Moreover, if the computer can handle 10^{10} primes per second, the calculation would take around 3×10^{130} years. (It's been suggested that you could go sit on the beach for 20 years, then buy a computer that is 1000 times faster, which would cut the runtime down to 3×10^{127} years – a very large savings!) Clearly, better methods are needed. Some of these are discussed in this section.

A very basic idea, one that is behind many factorization methods, is the following.

Basic Principle

Let n be an integer and suppose there exist integers x and y with $x^2 \equiv y^2 \pmod{n}$, but $x \not\equiv \pm y \pmod{n}$. Then n is composite. Moreover, $\gcd(x - y, n)$ gives a nontrivial factor of n .

Proof. Let $d = \gcd(x - y, n)$. If $d = n$ then $x \equiv y \pmod{n}$, which is assumed not to happen. Suppose $d = 1$. The Proposition in Subsection 3.3.1 says that if $ab \equiv ac \pmod{n}$ and if $\gcd(a, n) = 1$, then $b \equiv c \pmod{n}$. In our case, let $a = x - y$, let $b = x + y$, and let $c = 0$. Then $ab = x^2 - y^2 \equiv 0 \equiv ac$. If $d = \gcd(x - y, n) = 1$, then $x + y = b \equiv c = 0$. This says that

$x \equiv -y \pmod{n}$, which contradicts the assumption that $x \not\equiv -y \pmod{n}$. Therefore, $d \neq 1, n$, so d is a nontrivial factor of n .

Example

Since $12^2 \equiv 2^2 \pmod{35}$, but $12 \not\equiv \pm 2 \pmod{35}$, we know that 35 is composite. Moreover, $\gcd(12 - 2, 35) = 5$ is a nontrivial factor of 35.

It might be surprising, but factorization and primality testing are not the same. It is much easier to prove a number is composite than it is to factor it. There are many large integers that are known to be composite but that have not been factored. How can this be done? We give a simple example. We know by Fermat's theorem that if p is prime, then $2^{p-1} \equiv 1 \pmod{p}$. Let's use this to show 35 is not prime. By successive squaring, we find (congruences are mod 35)

$$\begin{aligned} 2^4 &\equiv 16, \\ 2^8 &\equiv 256 \equiv 11 \\ 2^{16} &\equiv 121 \equiv 16 \\ 2^{32} &\equiv 256 \equiv 11. \end{aligned}$$

Therefore,

$$2^{34} \equiv 2^{32}2^2 \equiv 11 \cdot 4 \equiv 9 \not\equiv 1 \pmod{35}.$$

Fermat's theorem says that 35 cannot be prime, so we have proved 35 to be composite without finding a factor.

The same reasoning gives us the following.

Fermat Primality Test

Let $n > 1$ be an integer. Choose a random integer a with $1 < a < n - 1$. If $a^{n-1} \not\equiv 1 \pmod{n}$, then n

is composite. If $a^{n-1} \equiv 1 \pmod{n}$, then n is probably prime.

Although this and similar tests are usually called “primality tests,” they are actually “compositeness tests,” since they give a completely certain answer only in the case when n is composite. The Fermat test is quite accurate for large n . If it declares a number to be composite, then this is guaranteed to be true. If it declares a number to be probably prime, then empirical results show that this is very likely true. Moreover, since modular exponentiation is fast, the Fermat test can be carried out quickly.

Recall that modular exponentiation is accomplished by successive squaring. If we are careful about how we do this successive squaring, the Fermat test can be combined with the Basic Principle to yield the following stronger result.

Miller-Rabin Primality Test

Let $n > 1$ be an odd integer. Write $n - 1 = 2^k m$ with m odd. Choose a random integer a with $1 < a < n - 1$. Compute $b_0 \equiv a^m \pmod{n}$. If $b_0 \equiv \pm 1 \pmod{n}$, then stop and declare that n is probably prime. Otherwise, let $b_1 \equiv b_0^2 \pmod{n}$. If $b_1 \equiv 1 \pmod{n}$, then n is composite (and $\gcd(b_0 - 1, n)$ gives a nontrivial factor of n). If $b_1 \equiv -1 \pmod{n}$, then stop and declare that n is probably prime. Otherwise, let $b_2 \equiv b_1^2 \pmod{n}$. If $b_2 \equiv 1 \pmod{n}$, then n is composite. If $b_2 \equiv -1 \pmod{n}$, then stop and declare that n is probably prime. Continue in this way until stopping or reaching b_{k-1} . If $b_{k-1} \not\equiv -1 \pmod{n}$, then n is composite.

Example

Let $n = 561$. Then $n - 1 = 560 = 16 \cdot 35$, so

$2^k = 2^4$ and $m = 35$. Let $a = 2$. Then

$$\begin{aligned} b_0 &\equiv 2^{35} \equiv 263 \pmod{561} \\ b_1 &\equiv b_0^2 \equiv 166 \pmod{561} \\ b_2 &\equiv b_1^2 \equiv 67 \pmod{561} \\ b_3 &\equiv b_2^2 \equiv 1 \pmod{561}. \end{aligned}$$

Since $b_3 \equiv 1 \pmod{561}$, we conclude that 561 is composite. Moreover, $\gcd(b_2 - 1, 561) = 33$, which is a nontrivial factor of 561.

If n is composite and $a^{n-1} \equiv 1 \pmod{n}$, then we say that n is a pseudoprime for the base a . If a and n are such that n passes the Miller-Rabin test, we say that n is a strong pseudoprime for the base a . We showed in [Section 3.6](#) that $2^{560} \equiv 1 \pmod{561}$, so 561 is a pseudoprime for the base 2. However, the preceding calculation shows that 561 is not a strong pseudoprime for the base 2. For a given base, strong pseudoprimes are much more rare than pseudoprimes.

Up to 10^{10} , there are 455052511 primes. There are 14884 pseudoprimes for the base 2, and 3291 strong pseudoprimes for the base 2. Therefore, calculating $2^{n-1} \pmod{n}$ will fail to recognize a composite in this range with probability less than 1 out of 30 thousand, and using the Miller-Rabin test with $a = 2$ will fail with probability less than 1 out of 100 thousand.

It can be shown that the probability that the Miller-Rabin test fails to recognize a composite for a randomly chosen a is at most $1/4$. In fact, it fails much less frequently than this. See [Damgård et al.]. If we repeat the test 10 times, say, with randomly chosen values of a , then we expect that the probability of certifying a composite number as prime is at most $(1/4)^{10} \simeq 10^{-6}$. In practice, using the test for a single a is fairly accurate.

Though strong pseudoprimes are rare, it has been proved (see [Alford et al.]) that, for any finite set B of bases, there are infinitely many integers that are strong pseudoprimes for all $b \in B$. The first strong pseudoprime for all the bases $b = 2, 3, 5, 7$ is 3215031751. There is a 337-digit number that is a strong pseudoprime for all bases that are primes < 200 .

Suppose we need to find a prime of around 300 digits. The prime number theorem asserts that the density of primes around x is approximately $1/\ln x$. When $x = 10^{300}$, this gives a density of around $1/\ln(10^{300}) = 1/690$. Since we can skip the even numbers, this can be raised to $1/345$. Pick a random starting point, and throw out the even numbers (and multiples of other small primes). Test each remaining number in succession by the Miller-Rabin test. This will tend to eliminate all the composites. On average, it will take less than 400 uses of the Miller-Rabin test to find a likely candidate for a prime, so this can be done fairly quickly. If we need to be completely certain that the number in question is prime, there are more sophisticated primality tests that can test a number of 300 digits in a few seconds.

Why does the test work? Suppose, for example, that $b_3 \equiv 1 \pmod{n}$. This means that $b_2^2 \equiv 1^2 \pmod{n}$. Apply the Basic Principle from before. Either $b_2 \equiv \pm 1 \pmod{n}$, or $b_2 \not\equiv \pm 1 \pmod{n}$ and n is composite. In the latter case, $\gcd(b_2 - 1, n)$ gives a nontrivial factor of n . In the former case, the algorithm would have stopped by the previous step. If we reach b_{k-1} , we have computed $b_{k-1} \equiv a^{(n-1)/2} \pmod{n}$. The square of this is a^{n-1} , which must be $1 \pmod{n}$ if n is prime, by Fermat's theorem. Therefore, if n is prime, $b_{k-1} \equiv \pm 1 \pmod{n}$. All other choices mean that n is composite. Moreover, if $b_{k-1} \equiv 1$, then, if we didn't stop at an earlier step, $b_{k-2}^2 \equiv 1^2 \pmod{n}$ with

$b_{k-2} \not\equiv \pm 1 \pmod{n}$. This means that n is composite (and we can factor n).

In practice, if n is composite, usually we reach b_{k-1} and it is not $\pm 1 \pmod{n}$. In fact, usually

$a^{n-1} \not\equiv 1 \pmod{n}$. This means that Fermat's test says n is not prime.

For example, let $n = 299$ and $a = 2$. Since

$2^{298} \equiv 140 \pmod{299}$, Fermat's theorem and also the Miller-Rabin test say that 299 is not prime (without factoring it). The reason this happens is the following.

Note that $299 = 13 \times 23$. An easy calculation shows that $2^{12} \equiv 1 \pmod{13}$ and no smaller exponent works. In fact, $2^j \equiv 1 \pmod{13}$ if and only if j is a multiple of 12. Since 298 is not a multiple of 12, we have

$2^{298} \not\equiv 1 \pmod{13}$, and therefore also

$2^{298} \not\equiv 1 \pmod{299}$. Similarly, $2^j \equiv 1 \pmod{23}$ if and only if j is a multiple of 11, from which we can again deduce that $2^{298} \not\equiv 1 \pmod{299}$. If Fermat's theorem (and the Miller-Rabin test) were to give us the wrong answer in this case, we would have needed

$13 \cdot 23 - 1$ to be a multiple of $12 \cdot 11$.

Consider the general case $n = pq$, a product of two primes. For simplicity, consider the case where $p > q$ and suppose $a^k \equiv 1 \pmod{p}$ if and only if $k \equiv 0 \pmod{p-1}$. This means that a is a primitive root mod p ; there are $\phi(p-1)$ such a mod p . Since $0 < q-1 < p-1$, we have

$$n-1 \equiv pq-1 \equiv q(p-1) + q-1 \not\equiv 0 \pmod{p-1}.$$

Therefore, $a^{n-1} \not\equiv 1 \pmod{p}$ by our choice of a , which implies that $a^{n-1} \not\equiv 1 \pmod{n}$. Similar reasoning shows that usually $a^{n-1} \not\equiv 1 \pmod{n}$ for many other choices of a , too.

But suppose we are in a case where $a^{n-1} \equiv 1 \pmod{n}$. What happens? Let's look at the example of $n = 561$.

Since $561 = 3 \times 11 \times 17$, we consider what is happening to the sequence $b_0, b_1, b_2, b_3 \pmod{3}$, $\pmod{11}$, and $\pmod{17}$:

$$\begin{aligned} b_0 &\equiv -1 \pmod{3}, \quad \equiv -1 \pmod{11}, \quad \equiv 8 \pmod{17}, \\ b_1 &\equiv 1 \pmod{3}, \quad \equiv 1 \pmod{11}, \quad \equiv -4 \pmod{17}, \\ b_2 &\equiv 1 \pmod{3}, \quad \equiv 1 \pmod{11}, \quad \equiv -1 \pmod{17}, \\ b_3 &\equiv 1 \pmod{3}, \quad \equiv 1 \pmod{11}, \quad \equiv 1 \pmod{17}. \end{aligned}$$

Since $b_3 \equiv 1 \pmod{561}$, we have $b_2^2 \equiv b_3 \equiv 1 \pmod{\text{all three primes}}$. But there is no reason that b_3 is the first time we get $b_i \equiv 1 \pmod{\text{a particular prime}}$. We already have $b_1 \equiv 1 \pmod{3}$ and $\pmod{11}$, but we have to wait for b_3 when working $\pmod{17}$. Therefore, $b_2^2 \equiv b_3 \equiv 1 \pmod{3, \pmod{11, \pmod{17}}}$, but b_2 is congruent to 1 only mod 3 and mod 11. Therefore, $b_2 - 1$ contains the factors 3 and 11, but not 17. This is why $\gcd(b_2 - 1, 561)$ finds the factor 33 of 561. The reason we could factor 561 by this method is that the sequence b_0, b_1, \dots reached 1 mod the primes not all at the same time.

More generally, consider the case $n = pq$ (a product of several primes is similar) and suppose

$a^{n-1} \equiv 1 \pmod{n}$. As pointed out previously, it is very unlikely that this is the case; but if it does happen, look at what is happening \pmod{p} and \pmod{q} . It is likely that the sequences $b_i \pmod{p}$ and $b_i \pmod{q}$ reach -1 and then 1 at different times, just as in the example of 561. In this case, we will have $b_i \equiv -1 \pmod{p}$ but $b_i \equiv 1 \pmod{q}$ for some i ; therefore, $b_i^2 \equiv 1 \pmod{n}$ but $b_i \not\equiv \pm 1 \pmod{n}$. Therefore, we'll be able to factor n .

The only way that n can pass the Miller-Rabin test is to have $a^{n-1} \equiv 1 \pmod{n}$ and also to have the sequences $b_i \pmod{p}$ and $b_i \pmod{q}$ reach 1 at the same time. This rarely happens.

Another primality test of a nature similar to the Miller-Rabin test is the following, which uses the Jacobi symbol (see [Section 3.10](#)).

Solovay-Strassen Primality Test

Let n be an odd integer. Choose several random integers a with $1 < a < n - 1$. If

$$\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod{n}$$

for some a , then n is composite. If

$$\left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod{n}$$

for all of these random a , then n is probably prime.

Note that if n is prime, then the test will declare n to be a probable prime. This is because of Part 2 of the second Proposition in [Section 3.10](#).

The Jacobi symbol can be evaluated quickly, as in [Section 3.10](#). The modular exponentiation can also be performed quickly.

For example,

$$\left(\frac{2}{15}\right) = -1 \not\equiv 23 \equiv 2^{(15-1)/2} \pmod{15},$$

so 15 is not prime. As in the Miller-Rabin tests, we usually do not get ± 1 for $a^{(n-1)/2} \pmod{n}$. Here is a case where it happens:

$$\left(\frac{2}{341}\right) = -1 \not\equiv +1 \equiv 2^{(341-1)/2} \pmod{341}.$$

However, the Solovay-Strassen test says that 341 is composite.

Both the Miller-Rabin and the Solovay-Strassen tests work quickly in practice, but, when p is prime, they do not give rigorous proofs that p is prime. There are tests that actually prove the primality of p , but they are somewhat slower and are used only when it is essential

that the number be proved to be prime. Most of these methods are probabilistic, in the sense that they work with very high probability in any given case, but success is not guaranteed. In 2002, Agrawal, Kayal, and Saxena [Agrawal et al.] gave what is known as a deterministic polynomial time algorithm for deciding whether or not a number is prime. This means that the computation time is always, rather than probably, bounded by a constant times a power of $\log p$. This was a great theoretical advance, but their algorithm has not yet been improved to the point that it competes with the probabilistic algorithms.

For more on primality testing and its history, see [Williams].

9.4 Factoring

We now turn to factoring. The basic method of dividing an integer n by all primes $p \leq \sqrt{n}$ is much too slow for most purposes. For many years, people have worked on developing more efficient algorithms. We present some of them here. In Chapter 21, we'll also cover a method using elliptic curves, and in Chapter 25, we'll show how a quantum computer, if built, could factor efficiently.

One method, which is also too slow, is usually called the **Fermat factorization** method. The idea is to express n as a difference of two squares: $n = x^2 - y^2$. Then $n = (x + y)(x - y)$ gives a factorization of n . For example, suppose we want to factor $n = 295927$. Compute $n + 1^2, n + 2^2, n + 3^2, \dots$, until we find a square. In this case, $295927 + 3^2 = 295936 = 544^2$. Therefore,

$$295927 = (544 + 3)(544 - 3) = 547 \cdot 541.$$

The Fermat method works well when n is the product of two primes that are very close together. If $n = pq$, it takes $|p - q|/2$ steps to find the factorization. But if p and q are two randomly selected 300-digit primes, it is likely that $|p - q|$ will be very large, probably around 300 digits, too. So Fermat factorization is unlikely to work. Just to be safe, however, the primes for an RSA modulus are often chosen to be of slightly different sizes.

We now turn to more modern methods. If one of the prime factors of n has a special property, it is sometimes easier to factor n . For example, if p divides n and $p - 1$ has only small prime factors, the following method is effective. It was invented by Pollard in 1974.

The $p - 1$ Factoring Algorithm

Choose an integer $a > 1$. Often $a = 2$ is used. Choose a bound B . Compute $b \equiv a^{B!} \pmod{n}$ as follows. Let $b_1 \equiv a \pmod{n}$ and $b_j \equiv b_{j-1}^j \pmod{n}$. Then $b_B \equiv b \pmod{n}$. Let $d = \gcd(b - 1, n)$. If $1 < d < n$, we have found a nontrivial factor of n .

Suppose p is a prime factor of n such that $p - 1$ has only small prime factors. Then it is likely that $p - 1$ will divide $B!$, say $B! = (p - 1)k$. By Fermat's theorem, $b \equiv a^{B!} \equiv (a^{p-1})^k \equiv 1 \pmod{p}$, so p will occur in the greatest common divisor of $b - 1$ and n . If q is another prime factor of n , it is unlikely that $b \equiv 1 \pmod{q}$, unless $q - 1$ also has only small prime factors. If $d = n$, not all is lost. In this case, we have an exponent r (namely $B!$) and an a such that $a^r \equiv 1 \pmod{n}$. There is a good chance that the $a^r \equiv 1$ method (explained later in this section) will factor n . Alternatively, we could choose a smaller value of B and repeat the calculation.

For an example, see Example 34 in the Computer Appendices.

How do we choose the bound B ? If we choose a small B , then the algorithm will run quickly but will have a very small chance of success. If we choose a very large B , then the algorithm will be very slow. The actual value used will depend on the situation at hand.

In the applications, we will use integers that are products of two primes, say $n = pq$, but that are hard to factor. Therefore, we should ensure that $p - 1$ has at least one large prime factor. This is easy to accomplish. Suppose we want p to have around 300 digits. Choose a large prime p_0 , perhaps around 10^{140} . Look at integers of the

form $kp_0 + 1$, with k running through some integers around 10^{160} . Test $kp_0 + 1$ for primality by the Miller-Rabin test, as before. On the average, this should produce a desired value of p in less than 400 steps. Now choose a large prime q_0 and follow the same procedure to obtain q . Then $n = pq$ will be hard to factor by the $p - 1$ method.

The elliptic curve factorization method (see [Section 21.3](#)) gives a generalization of the $p - 1$ method. However, it uses some random numbers near $p - 1$ and only requires at least one of them to have only small prime factors. This allows the method to detect many more primes p , not just those where $p - 1$ has only small prime factors.

9.4.1 $x^2 \equiv y^2$

Since it is the basis of the best current factorization methods, we repeat the following result from [Section 9.4](#).

Basic Principle

Let n be an integer and suppose there exist integers x and y with $x^2 \equiv y^2 \pmod{n}$, but $x \not\equiv \pm y \pmod{n}$. Then n is composite. Moreover, $\gcd(x - y, n)$ gives a nontrivial factor of n .

For an example, see Example 33 in the Computer Appendices.

How do we find the numbers x and y ? Let's suppose we want to factor $n = 3837523$. Observe the following:

$$\begin{aligned}
9398^2 &\equiv 5^5 \cdot 19 \pmod{3837523} \\
19095^2 &\equiv 2^2 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \pmod{3837523} \\
1964^2 &\equiv 3^2 \cdot 13^3 \pmod{3837523} \\
17078^2 &\equiv 2^6 \cdot 3^2 \cdot 11 \pmod{3837523}.
\end{aligned}$$

If we multiply the relations, we obtain

$$\begin{aligned}
(9398 \cdot 19095 \cdot 1964 \cdot 17078)^2 &\equiv (2^4 \cdot 3^2 \cdot 5^3 \cdot 11 \cdot 13^2 \cdot 19)^2 \\
2230387^2 &\equiv 2586705^2.
\end{aligned}$$

Since $2230387 \not\equiv \pm 2586705 \pmod{3837523}$, we now can factor 3837523 by calculating

$$\gcd(2230387 - 2586705, 3837523) = 1093.$$

The other factor is $3837523/1093 = 3511$.

Here is a way of looking at the calculations we just did.

First, we generate squares such that when they are reduced mod $n = 3837523$ they can be written as products of small primes (in the present case, primes less than 20). This set of primes is called our **factor base**. We'll discuss how to generate such squares shortly. Each of these squares gives a row in a matrix, where the entries are the exponents of the primes 2, 3, 5, 7, 11, 13, 17, 19. For example, the relation $17078^2 \equiv 2^6 \cdot 3^2 \cdot 11 \pmod{3837523}$ gives the row 6, 2, 0, 0, 1, 0, 0, 0.

In addition to the preceding relations, suppose that we have also found the following relations:

$$\begin{aligned}
8077^2 &\equiv 2 \cdot 19 \pmod{3837523} \\
3397^2 &\equiv 2^5 \cdot 5 \cdot 13^2 \pmod{3837523} \\
14262^2 &\equiv 5^2 \cdot 7^2 \cdot 13 \pmod{3837523}.
\end{aligned}$$

We obtain the matrix

9398	0	0	5	0	0	0	0	1
19095	2	0	1	0	1	1	0	1
1964	0	2	0	0	0	3	0	0
17078	6	2	0	0	1	0	0	0
8077	1	0	0	0	0	0	0	1
3397	5	0	1	0	0	2	0	0
14262	0	0	2	2	0	1	0	0

Now look for linear dependencies mod 2 among the rows. Here are three of them:

1. 1st + 5th + 6th = (6,0,6,0,0,2,0,2) $\equiv 0 \pmod{2}$
2. 1st + 2nd + 3rd + 4th = (8,4,6,0,2,4,0,2) $\equiv 0 \pmod{2}$
3. 3rd + 7th = (0,2,2,2,0,4,0,0) $\equiv 0 \pmod{2}$

When we have such a dependency, the product of the numbers yields a square. For example, these three yield

1. $(9398 \cdot 8077 \cdot 3397)^2 \equiv 2^6 \cdot 5^6 \cdot 13^2 \cdot 19^2 \equiv (2^3 \cdot 5^3 \cdot 13 \cdot 19)^2$
2. $(9398 \cdot 19095 \cdot 1964 \cdot 17078)^2 \equiv (2^3 \cdot 3^2 \cdot 5^3 \cdot 11 \cdot 13^2 \cdot 19)^2$
3. $(1964 \cdot 14262)^2 \equiv (3 \cdot 5 \cdot 7 \cdot 13^2)^2$

Therefore, we have $x^2 \equiv y^2 \pmod{n}$ for various values of x and y . If $x \not\equiv \pm y \pmod{n}$, then $\gcd(x - y, n)$ yields a nontrivial factor of n . If $x \equiv \pm y \pmod{n}$, then usually $\gcd(x - y, n) = 1$ or n , so we don't obtain a factorization. In our three examples, we have

1. $3590523^2 \equiv 247000^2$, but
 $3590523 \equiv -247000 \pmod{3837523}$
2. $2230387^2 \equiv 2586705^2$ and
 $\gcd(2230387 - 2586705, 3837523) = 1093$
3. $1147907^2 \equiv 17745^2$ and
 $\gcd(1147907 - 17745, 3837523) = 1093$

We now return to the basic question: How do we find the numbers 9398, 19095, etc.? The idea is to produce

squares that are slightly larger than a multiple of n , so they are small mod n . This means that there is a good chance they are products of small primes. An easy way is to look at numbers of the form $[\sqrt{in} + j]$ for small j and for various values of i . Here $[x]$ denotes the greatest integer less than or equal to x . The square of such a number is approximately $in + 2j\sqrt{in} + j^2$, which is approximately $2j\sqrt{in} + j^2 \pmod{n}$. As long as i is not too large, this number is fairly small, hence there is a good chance it is a product of small primes.

In the preceding calculation, we have

$8077 = [\sqrt{17n} + 1]$ and $9398 = [\sqrt{23n} + 4]$, for example.

The method just used is the basis of many of the best current factorization methods. The main step is to produce congruence relations

$$x^2 \equiv \text{product of small primes}.$$

An improved version of the above method is called the quadratic sieve. A recent method, the number field sieve, uses more sophisticated techniques to produce such relations and is somewhat faster in many situations. See [Pomerance] for a description of these two methods and for a discussion of the history of factorization methods.

See also [Exercise 52](#).

Once we have several congruence relations, they are put into a matrix, as before. If we have more rows than columns in the matrix, we are guaranteed to have a linear dependence relation mod 2 among the rows. This leads to a congruence $x^2 \equiv y^2 \pmod{n}$. Of course, as in the case of 1st + 5th + 6th $\equiv 0 \pmod{2}$ considered previously, we might end up with $x \equiv \pm y$, in which case we don't obtain a factorization. But this situation is expected to occur at most half the time. So if we have enough relations – for example, if there are several more rows than columns – then we should have a relation that

yields $x^2 \equiv y^2$ with $x \not\equiv \pm y$. In this case $\gcd(x - y, n)$ is a nontrivial factor of n .

In the last half of the twentieth century, there was dramatic progress in factoring. This was partly due to the development of computers and partly due to improved algorithms. A major impetus was provided by the use of factoring in cryptology, especially the RSA algorithm. Table 9.2 gives the factorization records (in terms of the number of decimal digits) for various years.

Table 9.2 Factorization Records

Year	Number of Digits
1964	20
1974	45
1984	71
1994	129
1999	155
2003	174
2005	200
2009	232

Table 9.2 Full Alternative Text

9.4.2 Using $a^r \equiv 1$

On the surface, the Miller-Rabin test looks like it might factor n quite often; but what usually happens is that b_{k-1} is reached without ever having $b_u \equiv \pm 1 \pmod{n}$. The problem is that usually $a^{n-1} \not\equiv 1 \pmod{n}$. Suppose, on the other hand, that we have some exponent

r , maybe not $n - 1$, such that $a^r \equiv 1 \pmod{n}$ for some a with $\gcd(a, n) = 1$. Then it is often possible to factor n .

$a^r \equiv 1$ Factorization Method

Suppose we have an exponent $r > 0$ and an integer a such that $a^r \equiv 1 \pmod{n}$. Write $r = 2^k m$ with m odd. Let $b_0 \equiv a^m \pmod{n}$, and successively define $b_{u+1} \equiv b_u^2 \pmod{n}$ for $0 \leq u \leq k - 1$. If $b_0 \equiv 1 \pmod{n}$, then stop; the procedure has failed to factor n . If, for some u , we have $b_u \equiv -1 \pmod{n}$, stop; the procedure has failed to factor n . If, for some u , we have $b_{u+1} \equiv 1 \pmod{n}$ but $b_u \not\equiv \pm 1 \pmod{n}$, then $\gcd(b_u - 1, n)$ gives a nontrivial factor of n .

Of course, if we take $a = 1$, then any r works. But then $b_0 = 1$, so the method fails. But if a and r are found by some reasonably sensible method, there is a good chance that this method will factor n .

This looks very similar to the Miller-Rabin test. The difference is that the existence of r guarantees that we have $b_{u+1} \equiv 1 \pmod{n}$ for some u , which doesn't happen as often in the Miller-Rabin situation. Trying a few values of a has a very high probability of factoring n .

Of course, we might ask how we can find an exponent r . Generally, this seems to be very difficult, and this test cannot be used in practice. However, it is useful in showing that knowing the decryption exponent in the RSA algorithm allows us to factor the modulus. Moreover, if a quantum computer is built, it will perform factorizations by finding such an exponent r via its unique quantum properties. See Chapter 25.

For an example for how this method is used in analyzing RSA, see Example 32 in the Computer Appendices.

9.5 The RSA Challenge

When the RSA algorithm was first made public in 1977, Rivest, Shamir, and Adleman made the following challenge.

Let the RSA modulus be

$n =$
11438162575788867669235779976146612010218296721242362
562561842935706935245733897830597123563958705058989075
147599290026879543541

and let $e = 9007$ be the encryption exponent. The ciphertext is

$c =$
968696137546220614771409222543558829057599911245743198
746951209308162982251457083569314766228839896280133919
90551829945157815154.

Find the message.

The only known way of finding the plaintext is to factor n . In 1977, it was estimated that the then-current factorization methods would take 4×10^{16} years to do this, so the authors felt safe in offering \$100 to anyone who could decipher the message before April 1, 1982. However, techniques have improved, and in 1994, Atkins, Graff, Lenstra, and Leyland succeeded in factoring n .

They used 524339 “small” primes, namely those less than 16333610, plus they allowed factorizations to include up to two “large” primes between 16333610 and 2^{30} . The idea of allowing large primes is the following: If one large prime q appears in two different relations, these can be multiplied to produce a relation with q squared. Multiplying by $q^{-2} \pmod{n}$ yields a relation

involving only small primes. In the same way, if there are several relations, each with the same two large primes, a similar process yields a relation with only small primes. The “birthday paradox” (see [Section 12.1](#)) implies that there should be several cases where a large prime occurs in more than one relation.

Six hundred people, with a total of 1600 computers working in spare time, found congruence relations of the desired type. These were sent by e-mail to a central machine, which removed repetitions and stored the results in a large matrix. After seven months, they obtained a matrix with 524339 columns and 569466 rows. Fortunately, the matrix was sparse, in the sense that most of the entries of the matrix were 0s, so it could be stored efficiently. Gaussian elimination reduced the matrix to a nonsparse matrix with 188160 columns and 188614 rows. This took a little less than 12 hours. With another 45 hours of computation, they found 205 dependencies. The first three yielded the trivial factorization of n , but the fourth yielded the factors

$$p = \\ 349052951084765094914784961990389813341776463849338784 \\ 3990820577,$$

$$q = \\ 327691329932667095499619881908344614131776429679929425 \\ 39798288533.$$

Computing $9007^{-1} \pmod{(p-1)(q-1)}$ gave the decryption exponent

$$d = \\ 106698614368578024442868771328920154780709906633937862 \\ 801226224496631063125911774470873340168597462306553968 \\ 544513277109053606095.$$

Calculating $c^d \pmod{n}$ yielded the plaintext message

$$200805001301070903002315180419000118050019172105011309 \\ 190800151919090618010705,$$

which, when changed back to letters using
 $a = 01$, $b = 02$, \dots , blank = 00, yielded

the magic words are squeamish ossifrage

(a squeamish ossifrage is an overly sensitive hawk; the message was chosen so that no one could decrypt the message by guessing the plaintext and showing that it encrypted to the ciphertext). For more details of this factorization, see [Atkins et al.]. If you want to see how the decryption works once the factorization is known, see Example 28 in the Computer Appendices.

9.6 An Application to Treaty Verification

Countries A and B have signed a nuclear test ban treaty. Now each wants to make sure the other doesn't test any bombs. How, for example, is country A going to use seismic data to monitor country B? Country A wants to put sensors in B, which then send data back to A. Two problems arise.

1. Country A wants to be sure that Country B doesn't modify the data.
2. Country B wants to look at the message before it's sent to be sure that nothing else, such as espionage data, is being transmitted.

These seemingly contradictory requirements can be met by reversing RSA. First, A chooses $n = pq$ to be the product of two large primes and chooses encryption and decryption exponents e and d . The numbers n and e are given to B, but p , q , and d are kept secret. The sensor (it's buried deep in the ground and is assumed to be tamper proof) collects the data x and uses d to encrypt x to $y \equiv x^d \pmod{n}$. Both x and y are sent first to country B, which checks that $y^e \equiv x \pmod{n}$. If so, it knows that the encrypted message y corresponds to the data x , and forwards the pair x, y to A. Country A then checks that $y^e \equiv x \pmod{n}$, also. If so, A can be sure that the number x has not been modified, since if x is chosen, then solving $y^e \equiv x \pmod{n}$ for y is the same as decrypting the RSA message x , and this is believed to be hard to do. Of course, B could choose a number y first, then let $x \equiv y^e \pmod{n}$, but then x would probably not be a meaningful message, so A would realize that something had been changed.

The preceding method is essentially the RSA signature scheme, which will be studied in [Section 13.1](#).

9.7 The Public Key Concept

In 1976, Diffie and Hellman described the concept of public key cryptography, though at that time no realizations of the concept were publicly known (as mentioned in the introduction to this chapter, Clifford Cocks of the British cryptographic agency CESG had invented a secret version of RSA in 1973). In this section, we give the general theory of public key systems.

There are several implementations of public key cryptography other than RSA. In later chapters we describe three of them. One is due to ElGamal and is based on the difficulty of finding discrete logarithms. A second is NTRU and involves lattice methods. The third is due to McEliece and uses error correcting codes. There are also public key systems based on the knapsack problem. We don't cover them in this book; some versions have been broken and they are generally suspected to be weaker than systems such as RSA and ElGamal.

A **public key cryptosystem** is built up of several components. First, there is the set M of possible messages (potential plaintexts and ciphertexts). There is also the set K of “keys.” These are not exactly the encryption/decryption keys; in RSA, a key k is a triple (e, d, n) with $ed \equiv 1 \pmod{\phi(n)}$. For each key k , there is an encryption function E_k and a decryption function D_k . Usually, E_k and D_k are assumed to map M to M , though it would be possible to have variations that allow the plaintexts and ciphertexts to come from different sets. These components must satisfy the following requirements:

1. $E_k(D_k(m)) = m$ and $D_k(E_k(m)) = m$ for every $m \in M$ and every $k \in K$.

2. For every m and every k , the values of $E_k(m)$ and $D_k(m)$ are easy to compute.
3. For almost every $k \in K$, if someone knows only the function E_k , it is computationally infeasible to find an algorithm to compute D_k .
4. Given $k \in K$, it is easy to find the functions E_k and D_k .

Requirement (1) says that encryption and decryption cancel each other. Requirement (2) is needed; otherwise, efficient encryption and decryption would not be possible. Because of (4), a user can choose a secret random k from K and obtain functions E_k and D_k . Requirement (3) is what makes the system public key. Since it is difficult to determine D_k from E_k , it is possible to publish E_k without compromising the security of the system.

Let's see how RSA satisfies these requirements. The message space can be taken to be all nonnegative integers. As we mentioned previously, a key for RSA is a triple $k = (e, d, n)$. The encryption function is

$$E_k(m) = m^e \pmod{n},$$

where we break m into blocks if $m \geq n$. The decryption function is

$$D_k(m) = m^d \pmod{n},$$

again with m broken into blocks if needed. The functions E_k and D_k are immediately determined from knowledge of k (requirement (4)) and are easy to compute (requirement (2)). They are inverses of each other since $ed \equiv 1 \pmod{\phi(n)}$, so (1) is satisfied. If we know E_k , which means we know e and n , then we have seen that it is (probably) computationally infeasible to determine d , hence D_k . Therefore, (3) is (probably) satisfied.

Once a public key system is set up, each user generates a key k and determines E_k and D_k . The encryption function E_k is made public, while D_k is kept secret. If

there is a problem with impostors, a trusted authority can be used to distribute and verify keys.

In a symmetric system, Bob can be sure that a message that decrypts successfully must have come from Alice (who could really be a group of authorized users) or someone who has Alice's key. Only Alice has been given the key, so no one else could produce the ciphertext. However, Alice could deny sending the message since Bob could have simply encrypted the message himself. Therefore, authentication is easy (Bob knows that the message came from Alice, if he didn't forge it himself) but non-repudiation is not (see [Section 1.2](#)).

In a public key system, anyone can encrypt a message and send it to Bob, so he will have no idea where it came from. He certainly won't be able to prove it came from Alice. Therefore, more steps are needed for authentication and non-repudiation. However, these goals are easily accomplished as follows.

Alice starts with her message m and computes $E_{k_b}(D_{k_a}(m))$, where k_a is Alice's key and k_b is Bob's key. Then Bob can decrypt using D_{k_b} to obtain $D_{k_a}(m)$. He uses the publicly available E_{k_a} to obtain $E_{k_a}(D_{k_a}(m)) = m$. Bob knows that the message must have come from Alice since no one else could have computed $D_{k_a}(m)$. For the same reason, Alice cannot deny sending the message. Of course, all this assumes that most random "messages" are meaningless, so it is unlikely that a random string of symbols decrypts to a meaningful message unless the string was the encryption of something meaningful.

It is possible to use one-way functions with certain properties to construct a public key cryptosystem. Let $f(m)$ be an invertible one-way function. This means $f(x)$ is easy to compute, but, given y , it is computationally infeasible to find the unique value of x

such that $y = f(x)$. Now suppose $f(x)$ has a **trapdoor**, which means that there is an easy way to solve $y = f(x)$ for x , but only with some extra information known only to the designer of the function. Moreover, it should be computationally infeasible for someone other than the designer of the function to determine this trapdoor information. If there is a very large family of one-way functions with trapdoors, they can be used to form a public key cryptosystem. Each user generates a function from the family in such a way that only that user knows the trapdoor. The user's function is then published as a public encryption algorithm. When Alice wants to send a message m to Bob, she looks up his function $f_b(x)$ and computes $y = f_b(m)$. Alice sends y to Bob. Since Bob knows the trapdoor for $f_b(x)$, he can solve $y = f_b(m)$ and thus find m .

In RSA, the functions $f(x) = x^e \pmod{n}$, for appropriate n and e , form the family of one-way functions. The secret trapdoor information is the factorization of n , or, equivalently, the exponent d . In the ElGamal system (Section 10.5), the one-way function is obtained from exponentiation modulo a prime, and the trapdoor information is knowledge of a discrete log. In NTRU (Section 23.4), the trapdoor information is a pair of small polynomials. In the McEliece system (Section 24.10), the trapdoor information is an efficient way for finding the nearest codeword (“error correction”) for certain linear binary codes.

9.8 Exercises

1. The ciphertext 5859 was obtained from the RSA algorithm using $n = 11413$ and $e = 7467$. Using the factorization $11413 = 101 \cdot 113$, find the plaintext.
2. Bob sets up a budget RSA cryptosystem. He chooses $p = 23$ and $q = 19$ and computes $n = 437 = pq$. He chooses the encryption exponent to be $e = 397$. Alice sends Bob the ciphertext $c = 123$. What is the plaintext? (You know p , q , and e).
3. Suppose your RSA modulus is $n = 55 = 5 \times 11$ and your encryption exponent is $e = 3$.
 1. Find the decryption exponent d .
 2. Assume that $\gcd(m, 55) = 1$. Show that if $c \equiv m^3 \pmod{55}$ is the ciphertext, then the plaintext is $m \equiv c^d \pmod{55}$. Do not quote the fact that RSA decryption works. That is what you are showing in this specific case.
4. Bob's RSA modulus is $979 = 11 \times 89$ and his encryption exponent is $e = 587$. Alice sends him the ciphertext $c = 10$. What is the plaintext?
5. The ciphertext 62 was obtained using RSA with $n = 667$ and $e = 3$. You know that the plaintext is either 9 or 10. Determine which it is without factoring n .
6. Alice and Bob are trying to use RSA, but Bob knows only one large prime, namely $p = 1093$. He sends $n = p = 1093$ and $e = 361$ to Alice. She encrypts her message m as $c \equiv m^e \pmod{n}$. Eve intercepts c and decrypts using the congruence $m \equiv c^d \pmod{n}$. What value of d should Eve use? Your answer should be an actual number. You may assume that Eve knows n and e , and she knows that n is prime.
7. Suppose you encrypt messages m by computing $c \equiv m^3 \pmod{101}$. How do you decrypt? (That is, you want a decryption exponent d such that $c^d \equiv m \pmod{101}$; note that 101 is prime.)
8. Bob knows that if an RSA modulus can be factored, then the system has bad security. Therefore, he chooses a modulus that cannot be factored, namely the prime $n = 131303$. He chooses his encryption exponent to be $e = 13$, and he encrypts a message

m as $c \equiv m^{13} \pmod{n}$. The decryption method is to compute $m \equiv c^d \pmod{n}$ for some d . (Hint: Fermat's theorem)

9. Let p be a large prime. Suppose you encrypt a message x by computing $y \equiv x^e \pmod{p}$ for some (suitably chosen) encryption exponent e . How do you find a decryption exponent d such that $y^d \equiv x \pmod{p}$?
10. Bob decides to test his new RSA cryptosystem. He has RSA modulus $n = pq$ and encryption exponent e . His message is m . He sends $c \equiv m^e \pmod{n}$ to himself. Then, just for fun, he also sends $c' \equiv (m+q)^e \pmod{n}$ to himself. Eve knows n and e , and intercepts both c and c' . She guesses what Bob has done. How can she find the factorization of n ? (Hint: Show that $c \equiv c' \pmod{q}$ but not mod p . What is $\gcd(c - c', n)$?)
11. Let n be the product of two large primes. Alice wants to send a message m to Bob, where $\gcd(m, n) = 1$. Alice and Bob choose integers a and b relatively prime to $\phi(n)$. Alice computes $c \equiv m^a \pmod{n}$ and sends c to Bob. Bob computes $d \equiv c^b \pmod{n}$ and sends d back to Alice. Since Alice knows a , she finds a_1 such that $aa_1 \equiv 1 \pmod{\phi(n)}$. Then she computes $e \equiv d^{a_1} \pmod{n}$ and sends e to Bob. Explain what Bob must now do to obtain m , and show that this works. (Remark: In this protocol, the prime factors of n do not need to be kept secret. Instead, the security depends on keeping a, b secret. The present protocol is a less efficient version of the three-pass protocol from [Section 3.5](#).)
12. A bank in Alice Springs (Australia), also known as Alice, wants to send a lot of financial data to the Bank of Baltimore, also known as Bob. They want to use AES, but they do not share a common key. All of their communications will be on public airwaves. Describe how Alice and Bob can accomplish this using RSA.
13. Naive Nelson uses RSA to receive a single ciphertext c , corresponding to the message m . His public modulus is n and his public encryption exponent is e . Since he feels guilty that his system was used only once, he agrees to decrypt any ciphertext that someone sends him, as long as it is not c , and return the answer to that person. Evil Eve sends him the ciphertext $2^e c \pmod{n}$. Show how this allows Eve to find m .
14. Eve loves to do double encryption. She starts with a message m . First, she encrypts it twice with a one-time pad (the same one each time). Then she encrypts the result twice using a Vigenère cipher with key *NANANA*. Finally, she encrypts twice with RSA using modulus $n = pq = 7919 \times 17389$ and exponent $e = 66909025$. It happens that $e^2 \equiv 1 \pmod{(p-1)(q-1)}$. Show that the final result of all this encryption is the original plaintext. Explain your answer fully. Simply saying something like “decryption is the same as encryption” is not enough. You must explain why.

15. In order to increase security, Bob chooses n and two encryption exponents e_1, e_2 . He asks Alice to encrypt her message m to him by first computing $c_1 \equiv m^{e_1} \pmod{n}$, then encrypting c_1 to get $c_2 \equiv c_1^{e_2} \pmod{n}$. Alice then sends c_2 to Bob. Does this double encryption increase security over single encryption? Why or why not?

16. 1. Eve thinks that she has a great strategy for breaking RSA that uses a modulus n that is the product of two 300-digit primes. She decides to make a list of all 300-digit primes and then divide each of them into n until she factors n . Why won't this strategy work?

2. Eve has another strategy. She will make a list of all m with $0 \leq m < 10^{600}$, encrypt each m , and store them in a database. Suppose Eve has a superfast computer that can encrypt 10^{50} plaintexts per second (this is, of course, well beyond the speed of any existing technology). How many years will it take Eve to compute all 10^{600} encryptions? (There are approximately 3×10^7 seconds in a year.)

17. The exponents $e = 1$ and $e = 2$ should not be used in RSA. Why?

18. Alice is trying to factor $n = 57677$. She notices that $1234^4 \equiv 1 \pmod{n}$ and that $1234^2 \equiv 23154 \pmod{n}$. How does she use this information to factor n ? Describe the steps but do not actually factor n .

19. Let p and q be distinct odd primes, and let $n = pq$. Suppose that the integer x satisfies $\gcd(x, pq) = 1$.

1. Show that $x^{\frac{1}{2}\phi(n)} \equiv 1 \pmod{p}$ and

$$x^{\frac{1}{2}\phi(n)} \equiv 1 \pmod{q}.$$

2. Use (a) to show that $x^{\frac{1}{2}\phi(n)} \equiv 1 \pmod{n}$.

3. Use (b) to show that if $ed \equiv 1 \pmod{\frac{1}{2}\phi(n)}$ then

$x^{ed} \equiv x \pmod{n}$. (This shows that we could work with $\frac{1}{2}\phi(n)$ instead of $\phi(n)$ in RSA. In fact, we could also use the least common multiple of $p - 1$ and $q - 1$ in place of $\phi(n)$, by similar reasoning.)

20. Alice uses RSA with $n = 27046456501$ and $e = 3$. Her ciphertext is $c = 1860867$. Eve notices that $c^{14} \equiv 1 \pmod{n}$.

1. Show that $m^{14} \equiv 1 \pmod{n}$, where m is the plaintext.

2. Explicitly find an exponent f such that

$$c^f \equiv m \pmod{n}. \text{ (Hint: You do not need to factor } n \text{ to}$$

find f . Look at the proof that RSA decryption works. The only property of $\phi(n)$ that is used is that $m^{\phi(n)} \equiv 1$.)

21. Suppose that there are two users on a network. Let their RSA moduli be n_1 and n_2 , with n_1 not equal to n_2 . If you are told that n_1 and n_2 are not relatively prime, how would you break their systems?
22. Huey, Dewey, and Louie ask their uncle Donald, “Is $n = 19887974881$ prime or composite?” Donald replies, “Yes.” Therefore, they decide to obtain more information on their own.

1. Huey computes $13^{(n-1)} \equiv 16739180549 \pmod{n}$.
What does he conclude?

2. Dewey computes $7^{n-1} \equiv 1 \pmod{n}$, and he does this by computing

$$\begin{aligned} 7^{(n-1)/32} &\equiv 1992941816 \pmod{n} \\ 7^{(n-1)/16} &\equiv 19887730619 \\ 7^{(n-1)/8} &\equiv 1 \\ 7^{(n-1)/4} &\equiv 7^{(n-1)/2} \equiv 7^{(n-1)} \equiv 1. \end{aligned}$$

What information can Dewey obtain from his calculation that Huey does not obtain?

3. Louie notices that $19857930655^2 \equiv 123^2 \pmod{n}$.
What information can Louie compute? (In parts (b) and (c), you do not need to do the calculations, but you should indicate what calculations need to be done.)

23. You are trying to factor $n = 642401$. Suppose you discover that

$$516107^2 \equiv 7 \pmod{n}$$

and that

$$187722^2 \equiv 2^2 \cdot 7 \pmod{n}.$$

Use this information to factor n .

24. Suppose you know that $7961^2 \equiv 7^2 \pmod{8051}$. Use this information to factor 8051.

25. Suppose you discover that

$$\begin{aligned} 880525^2 &\equiv 2, & 2057202^2 &\equiv 3, & 648581^2 &\equiv 6, \\ 668676^2 &\equiv 77 \pmod{2288233}. \end{aligned}$$

How would you use this information to factor 2288233? Explain what steps you would do, but do not perform the numerical calculations.

26. Suppose you want to factor an integer n . You have found some integers x_1, x_2, x_3, x_4 such that

$$\begin{aligned}x_1^2 &\equiv 2 \cdot 3 \cdot 7 \pmod{n}, & x_2^2 &\equiv 3 \cdot 5 \cdot 7 \pmod{n} \\x_3^2 &\equiv 3^9 \pmod{n}, & x_4^2 &\equiv 2 \cdot 7 \pmod{n}.\end{aligned}$$

Describe how you might be able to use this information to factor n ? Why might the method fail?

27. Suppose you have two distinct large primes p and q . Explain how you can find an integer x such that

$$x^2 \equiv 49 \pmod{pq}, \quad x \not\equiv \pm 7 \pmod{pq}.$$

(Hint: Use the Chinese Remainder Theorem to find four solutions to $x^2 \equiv 49 \pmod{pq}$.)

28. You are told that

$$5^{945} \equiv 1768 \pmod{1891}, \quad 5^{1890} \equiv 1 \pmod{1891}.$$

Use this information to factor 1891.

29. Suppose n is a large odd number. You calculate

$2^{(n-1)/2} \equiv k \pmod{n}$, where k is some integer with $k \not\equiv \pm 1 \pmod{n}$.

1. Suppose $k^2 \not\equiv 1 \pmod{n}$. Explain why this implies that n is not prime.
2. Suppose $k^2 \equiv 1 \pmod{n}$. Explain how you can use this information to factor n .

- 30.
1. Bob is trying to set up an RSA cryptosystem. He chooses $n = pq$ and e , as usual. By encrypting messages six times, Eve guesses that $e^6 \equiv 1 \pmod{(p-1)(q-1)}$. If this is the case, what is the decryption exponent d ? (That is, give a formula for d in terms of the parameters n and e that allows Eve to compute d .)

2. Bob tries again, with a new n and e . Alice computes $c \equiv m^e \pmod{n}$. Eve sees no way to guess the decryption exponent d this time. Knowing that if she finds d she will have to do modular exponentiation, Eve starts computing successive squares:
 $c, c^2, c^4, c^8, \dots \pmod{n}$. She notices that $c^{32} \equiv 1 \pmod{n}$ and realizes that this means that $m^{32} \equiv 1 \pmod{n}$. If $e = 53$, what is a value for d that will decrypt the ciphertext? Prove that this value works.

31. Suppose two users Alice and Bob have the same RSA modulus n and suppose that their encryption exponents e_A and e_B are relatively prime. Charles wants to send the message m to Alice and Bob, so he encrypts to get $c_A \equiv m^{e_A}$ and

$c_B \equiv m^{e_B} \pmod{n}$. Show how Eve can find m if she intercepts c_A and c_B .

32. Bob finally sets up a very secure RSA system. In fact, it is so secure that he decides to tell Alice one of the prime factors of n ; call it p . Being no dummy, he does not take the message as p , but instead uses a shift cipher to hide the prime in the plaintext $m = p + 1$, and then does an RSA encryption to obtain $c \equiv m^e \pmod{n}$. He then sends c to Alice. Eve intercepts c , n and e , and she knows that Bob has encrypted p this way. Explain how she obtains p and q quickly. (Hint: How does c differ from the result of encrypting the simple plaintext “1”?)
33. Suppose Alice uses the RSA method as follows. She starts with a message consisting of several letters, and assigns $a = 1, b = 2, \dots, z = 26$. She then encrypts each letter separately. For example, if her message is *cat*, she calculates $3^e \pmod{n}$, $1^e \pmod{n}$, and $20^e \pmod{n}$. Then she sends the encrypted message to Bob. Explain how Eve can find the message without factoring n . In particular, suppose $n = 8881$ and $e = 13$. Eve intercepts the message

4461 794 2015 2015 3603.

Find the message without factoring 8881.

34. Let $n = 3837523$. Bob Square Messages sends and receives only messages m such that m is a square mod n and $\gcd(m, n) = 1$. It can be shown that $m^{958230} \equiv 1 \pmod{n}$ for such messages (even though $\phi(n) \neq 958230$). Bob chooses d and e satisfying $de \equiv 1 \pmod{958230}$. Show that if Alice sends Bob a ciphertext $c \equiv m^e \pmod{n}$ (where m is a square mod n , and $\gcd(m, n) = 1$), then Bob can decrypt by computing $c^d \pmod{n}$. Explain your reasoning.
35. Show that if $x^2 \equiv y^2 \pmod{n}$ and $x \not\equiv \pm y \pmod{n}$, then $\gcd(x + y, n)$ is a nontrivial factor of n .
36. Bob’s RSA system uses $n = 2776099$ and $e = 421$. Alice encrypts the message 2718 and sends the ciphertext to Bob. Unfortunately (for Alice), $2718^{10} \equiv 1 \pmod{n}$. Show that Alice’s ciphertext is the same as the plaintext. (Do not factor n . Do not compute $m^{421} \pmod{n}$ without using the extra information that $2718^{10} \equiv 1$. Do not claim that $\phi(2776099 - 10) = 0$; it doesn’t.)

37. Let $n = pq$ be the product of two distinct primes.

1. Let m be a multiple of $\phi(n)$. Show that if $\gcd(a, n) = 1$, then $a^m \equiv 1 \pmod{p}$ and $\equiv 1 \pmod{q}$.
2. Suppose m is as in part (a), and let a be arbitrary (possibly $\gcd(a, n) \neq 1$). Show that $a^{m+1} \equiv a \pmod{p}$ and $\equiv a \pmod{q}$.

3. Let e and d be encryption and decryption exponents for RSA with modulus n . Show that $a^{ed} \equiv a \pmod{n}$ for all a . This shows that we do not need to assume $\gcd(a, n) = 1$ in order to use RSA.
4. If p and q are large, why is it likely that $\gcd(a, n) = 1$ for a randomly chosen a ?
38. Alice and Bob are celebrating Pi Day. Alice calculates $23039^2 \equiv 314 \pmod{79927}$ and Bob calculates $35118^2 \equiv 314 \pmod{79927}$.
1. Use this information to factor 79927. (You should show how to use the information. Do not do the calculation. The answer is not “Put the number in the computer and then relax for 10 minutes.”)
 2. Given that $79927 = 257 \times 311$ and that $166^2 \equiv 314 \pmod{257}$ and $25^2 \equiv 314 \pmod{311}$, how would you produce the numbers 23039 and 35118 in part (a)? You do not need to do the calculations, but you should state which congruences are being solved and what theorems are being used.
 3. Now that Eve knows that $79927 = 257 \times 311$, she wants to find an $x \not\equiv \pm 17 \pmod{79927}$ such that $x^2 \equiv 17^2 \pmod{79927}$. Explain how to accomplish this. Say what the method is. Do not do the calculation.
39. Suppose $n = pqr$ is the product of three distinct primes. How would an RSA-type scheme work in this case? In particular, what relation would e and d satisfy?
- Note: There does not seem to be any advantage in using three primes instead of two. The running times of some factorization methods depend on the size of the smallest prime factor. Therefore, if three primes are used, the size of n must be increased in order to achieve the same level of security as obtained with two primes.
40. Suppose Bob's public key is $n = 2181606148950875138077$ and he has $e = 7$ as his encryption exponent. Alice encrypts the message $hi\ eve = 080900052205 = m$. By chance, the message m satisfies $m^3 \equiv 1 \pmod{n}$. If Eve intercepts the ciphertext, how can Eve read the message without factoring n ?
41. Let $p = 7919$ and $q = 17389$. Let $e = 66909025$. A calculation shows that $e^2 \equiv 1 \pmod{(p-1)(q-1)}$. Alice decides to encrypt the message $m = 12345$ using RSA with modulus $n = pq$ and exponent e . Since she wants the encryption to be very secure, she encrypts the ciphertext, again using n and e (so she has double encrypted the original plaintext). What is the final

ciphertext that she sends? Justify your answer without using a calculator.

42. You are told that $7172^2 \equiv 60^2 \pmod{14351}$. Use this information to factor 14351. You must use this information and you must give all steps of the computation (that is, give the steps you use if you are doing it completely without a calculator).
43.
 1. Show that if $\gcd(e, 24) = 1$, then $e^2 \equiv 1 \pmod{24}$.
 2. Show that if $n = 35$ is used as an RSA modulus, then the encryption exponent e always equals the decryption exponent d .
44.
 1. The exponent $e = 3$ has sometimes been used for RSA because it makes encryption fast. Suppose that Alice is encrypting two messages, $m_0 = m$ and $m_1 = m + 1$ (for example, these could be two messages that contain a counter variable that increments). Eve does not know m but knows that the two plaintexts differ by 1. Let $c_0 \equiv m_0^3$ and $c_1 \equiv m_1^3 \pmod{n}$. Show that if Eve knows c_0 and c_1 , she can recover m . (Hint: Compute $(c_1 + 2c_0 - 1)/(c_1 - c_0 + 2)$.)
 2. Suppose that $m_0 = m$ and $m_1 = m + b$ for some publicly known b . Modify the technique of part (a) so that Eve can recover m from the ciphertexts c_0 and c_1 .
45. Your opponent uses RSA with $n = pq$ and encryption exponent e and encrypts a message m . This yields the ciphertext $c \equiv m^e \pmod{n}$. A spy tells you that, for this message, $m^{12345} \equiv 1 \pmod{n}$. Describe how to determine m . Note that you do not know p , q , $\phi(n)$, or the secret decryption exponent d . However, you should find a decryption exponent that works for this particular ciphertext. Moreover, explain carefully why your decryption works (your explanation must include how the spy's information is used). For simplicity, assume that $\gcd(12345, e) = 1$.
46.
 1. Show that if $\gcd(b, 1729) = 1$ then $b^{1728} \equiv 1 \pmod{1729}$. You may use the fact that $1729 = 7 \cdot 13 \cdot 19$.
 2. By part (a), you know that $2^{1728} \equiv 1 \pmod{1729}$. When checking this result, you compute $2^{54} \equiv 1065 \pmod{1729}$ and $2^{108} \equiv 1 \pmod{1729}$. Use this information to find a nontrivial factor of 1729.
47. Suppose you are using RSA (with modulus $n = pq$ and encrypting exponent e), but you decide to restrict your messages to numbers m satisfying $m^{1000} \equiv 1 \pmod{n}$.

1. Show that if d satisfies $de \equiv 1 \pmod{1000}$, then d works as a decryption exponent for these messages.

2. Assume that both p and q are congruent to 1 mod 1000.
Determine how many messages satisfy
 $m^{1000} \equiv 1 \pmod{n}$. You may assume and use the fact that $m^{1000} \equiv 1 \pmod{r}$ has 1000 solutions when r is a prime congruent to 1 mod 1000.

48. You may assume the fact that $m^{270300} \equiv 1 \pmod{1113121}$ for all m with $\gcd(m, 1113121) = 1$. Let e and d satisfy $ed \equiv 1 \pmod{270300}$, and suppose that m is a message such that $0 < m < 1113121$ and $\gcd(m, 1113121) = 1$. Encrypt m as $c \equiv m^e \pmod{1113121}$. Show that $m \equiv c^d \pmod{1113121}$. Show explicitly how you use the fact that $ed \equiv 1 \pmod{270300}$ and the fact that $m^{270300} \equiv 1 \pmod{1113121}$. (Note: $\phi(1113121) \neq 270300$, so Euler's theorem does not apply.)

49. Suppose Bob's encryption company produces two machines, A and B, both of which are supposed to be implementations of RSA using the same modulus $n = pq$ for some unknown primes p and q . Both machines also use the same encryption exponent e . Each machine receives a message m and outputs a ciphertext that is supposed to be $m^e \pmod{n}$. Machine A always produces the correct output c . However, Machine B, because of implementation and hardware errors, always outputs a ciphertext $c' \pmod{n}$ such that $c' \equiv m^e \pmod{p}$ and $c' \equiv m^e + 1 \pmod{q}$. How could you use machines A and B to find p and q ? (See Computer Problem 11 for a discussion of how such a situation could arise.) (Hint: $c \equiv c' \pmod{p}$ but not mod q . What is $\gcd(c - c', n)$?)

50. Alice and Bob play the following game. They choose a large odd integer n and write $n - 1 = 2^k m$ with m odd. Alice then chooses a random integer $r \not\equiv \pm 1 \pmod{n}$ with $\gcd(r, n) = 1$. Bob computes $x_1 \equiv r^m \pmod{n}$. Then Alice computes $x_2 \equiv x_1^2 \pmod{n}$. Then Bob computes $x_3 \equiv x_2^2 \pmod{n}$, Alice computes $x_4 \equiv x_3^2 \pmod{n}$, etc. They stop if someone gets $\pm 1 \pmod{n}$, and the person who gets ± 1 wins.

 1. Show that if n is prime, the game eventually stops.

 2. Suppose n is the product of two distinct primes and Alice knows this factorization. Show how Alice can choose r so that she wins on her first play. That is, $x_1 \not\equiv \pm 1 \pmod{n}$ but $x_2 \equiv \pm 1 \pmod{n}$.

51. 1. Suppose Alice wants to send a short message m but wants to prevent the short message attack of [Section 9.2](#). She tells Bob that she is adjoining 100 zeros at the end of her plaintext, so she is using $m_1 = 10^{100}m$ as the plaintext and sending $c_1 \equiv m_1^e$. If Eve knows that Alice

is doing this, how can Eve modify the short plaintext attack and possibly find the plaintext?

2. Suppose Alice realizes that the method of part (a) does not provide security, so instead she makes the plaintext longer by repeating it two times: $m \parallel m$ (where $x \parallel y$ means we write the digits of x followed by the digits of y to obtain a longer number). If Eve knows that Alice is doing this, how can Eve modify the short plaintext attack and possibly find the plaintext? Assume that Eve knows the length of m . (Hint: Express $m \parallel m$ as a multiple of m .)

52. This exercise provides some of the details of how the quadratic sieve obtains the relations that are used to factor a large odd integer n . Let s be the smallest integer greater than the square root of n and let $f(x) = (x + s)^2 - n$. Let the factor base B consist of the primes up to some bound B . We want to find squares that are congruent mod n to a product of primes in B . One way to do this is to find values of $f(x)$ that are products of primes in B . We'll search over a range $0 \leq x \leq A$, for some A .

1. Suppose $0 \leq x < (\sqrt{2} - 1)\sqrt{n} - 1$. Show that $0 \leq f(x) < n$, so $f(x) \pmod n$ is simply $f(x)$. (Hint: Show that $x + s < x + \sqrt{n} + 1 < \sqrt{2n}$.) Henceforth, we'll assume that $A < (\sqrt{2} - 1)\sqrt{n} - 1$, so the values of x that we consider have $f(x) < n$.
2. Let p be a prime in B . Show that if there exists an integer x with $f(x)$ divisible by p , then n is a square mod p . This shows that we may discard those primes in B for which n is not a square mod p . Henceforth, we will assume that such primes have been discarded.
3. Let $p \in B$ be such that n is a square mod p . Show that if p is odd, and $p \nmid n$, then there are exactly two values of $x \pmod p$ such that $f(x) \equiv 0 \pmod p$. Call these values $x_{p,1}$ and $x_{p,2}$. (Note: In the unlikely case that $p|n$, we have found a factor, which was the goal.)
4. For each x with $0 \leq x \leq A$, initialize a register with value $\log f(x)$. For each prime $p \in B$, subtract $\log p$ from the registers of those x with $x \equiv x_{p,1} \text{ or } x_{p,2} \pmod p$. (Remark: This is the “sieving” part of the quadratic sieve.) Show that if $f(x)$ (with $0 \leq x \leq A$) is a product of distinct primes in B , then the register for x becomes 0 at the end of this process.
5. Explain why it is likely that if $f(x)$ (with $0 \leq x \leq A$) is a product of (possibly nondistinct) primes in B , then the final result for the register for x is small (compared to

the register for an x such that $f(x)$ has a prime factor not in B).

6. Why is the procedure of part (d) faster than trial division of each $f(x)$ by each element of B , and why does the algorithm subtract $\log p$ rather than dividing $f(x)$ by p ?

In practice, the sieve also takes into account solutions to $f(x) \equiv 0 \pmod{\text{some powers of small primes in } B}$. After the sieving process is complete, the registers with small entries are checked to see which correspond to $f(x)$ being a product of primes from B . These give the relations “square \equiv product of primes in $B \pmod{n}$ ” that are used to factor n .

53. Bob chooses $n = pq$ to be the product of two large primes p, q such that $\gcd(pq, (p-1)(q-1)) = 1$.

1. Show that $\phi(n^2) = n\phi(n)$ (this is true for any positive n).

2. Let $k \geq 0$. Show that $(1+n)^k \equiv 1 + kn \pmod{n^2}$ (this is true for any positive n).

3. Alice has message represented as $m \pmod{n}$. She encrypts m by first choosing a random integer r with $\gcd(r, n) = 1$. She then computes

$$c \equiv (1+n)^m r^n \pmod{n^2}.$$

4. Bob decrypts by computing $m' \equiv c^{\phi(n)} \pmod{n^2}$.

Show that $m' \equiv 1 + \phi(n)m n \pmod{n^2}$. Therefore, Bob can recover the message by computing $(m' - 1)/n$ and then dividing by $\phi(n) \pmod{n}$.

5. Let $E(m)$ and $D(c)$ denote encryption and decryption via the method in parts (c) and (d). Show that

$$D(E(m_1)E(m_2)) \equiv D(E(m_1 + m_2)) \pmod{n}.$$

(9.1)

Note: The encryptions probably use different values of the random number r , so there is more than one possible encryption of a message m . Part (e) says that, no matter what choices are made for r , the decryption of $E(m_1)E(m_2)$ is the same as the decryption of $E(m_1 + m_2)$.

The preceding is called the **Paillier cryptosystem**. (Equation 9.1) says that it is possible to do addition on the encrypted messages without knowing the messages. For many years, a goal was to design a cryptosystem where both addition and multiplication could be done on the encrypted messages. This property is called **homomorphic encryption**, and the first such

system was designed by Gentry in 2009. Current research aims at designing systems that can be used in practice.

54. One possible application of the Paillier cryptosystem from the previous exercise is to electronic voting (but, as we'll see, modifications are needed in order to make it secure). Bob, who is the trusted authority, sets up the system. Each voter uses $m = 0$ for NO and $m = 1$ for YES. The voters encrypt their ms and send the ciphertexts to Bob.

1. How does Bob determine how many YES and how many NO votes without decrypting the individual votes?
2. Suppose an overzealous and not very honest voter wants to increase the number of YES votes. How is this accomplished?
3. Suppose someone else wants to increase the number of NO votes. How can this be done?

55. Here is a 3-person encryption scheme based on the same principles as RSA. A trusted entity chooses two large distinct primes p and q and computes $n = pq$, then chooses three integers k_1, k_2, k_3 with $k_1k_2k_3 \equiv 1 \pmod{(p-1)(q-1)}$. Alice, Bob, and Carla are given the following keys:

$$\text{Alice: } (n, k_1, k_2), \quad \text{Bob: } (n, k_2, k_3), \quad \text{Carla: } (n, k_1, k_3).$$

1. Alice has a message that she wants to send to both Bob and Carla. How can Alice encrypt the message so that both of them can read decrypt it?
2. Alice has a message that she wants to send only to Carla. How can Alice encrypt the message so that Carla can decrypt it but Bob cannot decrypt it?

9.9 Computer Problems

1. 1. Paul Revere's friend in a tower at MIT says he'll send the message *one* if (the British are coming) by land and *two* if by sea. Since they know that RSA will be invented in the Boston area, they decide that the message should be encrypted using RSA with $n = 712446816787$ and $e = 6551$. Paul Revere receives the ciphertext 273095689186. What was the plaintext? Answer this without factoring n .
2. What could Paul Revere's friend have done so that we couldn't guess which message was encrypted? (See the end of Subsection 9.2.2.)
2. In an RSA cryptosystem, suppose you know $n = 718548065973745507$, $e = 3449$, and $d = 543546506135745129$. Factor n using the $a^r \equiv 1$ method of Subsection 9.4.2.
3. Choose two 30-digit primes p and q and an encryption exponent e . Encrypt each of the plaintexts *cat*, *bat*, *hat*, *encyclopedia*, *antidisestablishmentarianism*. Can you tell from looking at the ciphertexts that the first three plaintexts differ in only one letter or that the last two plaintexts are much longer than the first three?
4. Factor 618240007109027021 by the $p - 1$ method.
5. Factor 8834884587090814646372459890377418962766907 by the $p - 1$ method. (The number is stored in the downloadable computer files (bit.ly/2JbcS6p) as *n1*.)
6. Let $n = 537069139875071$. Suppose you know that

$$85975324443166^2 \equiv 462436106261^2 \pmod{n}.$$

Factor n .

7. Let $n = 985739879 \cdot 1388749507$. Find x and y with $x^2 \equiv y^2 \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$.

8. 1. Suppose you know that

$$33335^2 \equiv 670705093^2 \pmod{670726081}.$$

Use this information to factor 670726081.

2. Suppose you know that $3^2 \equiv 670726078^2 \pmod{670726081}$. Why won't this

information help you to factor 670726081?

9. Suppose you know that

$$\begin{aligned} 2^{958230} &\equiv 1488665 \pmod{3837523} \\ 2^{1916460} &\equiv 1 \pmod{3837523}. \end{aligned}$$

How would you use this information to factor 3837523? Note that the exponent 1916460 is twice the exponent 958230.

Alice and Bob have the same RSA modulus n , given to them by some central authority (who does not tell them the factorization of n). Alice has encryption and decryption exponents e_A and d_A , and Bob has e_B and d_B . As usual, e_A and e_B are public and d_A and d_B are private.

- 10.
1. Suppose the primes p and q used in the RSA algorithm are consecutive primes. How would you factor $n = pq$?
 2. The ciphertext 10787770728 was encrypted using $e = 113$ and $n = 10993522499$. The factors p and q of n were chosen so that $q - p = 2$. Decrypt the message.
 3. The following ciphertext c was encrypted mod n using the exponent e :

$$\begin{aligned} n &= 152415787501905985701881832150835089037858868621211004433 \\ e &= 9007 \\ c &= 141077461765569500241199505617854673388398574333341423525. \end{aligned}$$

The prime factors p and q of n are consecutive primes. Decrypt the message. (The number n is stored in the downloadable computer files (bit.ly/2JbcS6p) as *naive*, and c is stored as *cnaive*.) Note: In Mathematica®, the command **Round[N[Sqrt[n],50]]** evaluates the square root of n to 50 decimal places and then rounds to the nearest integer. In Maple, first use the command **Digits:=50** to obtain 50-digit accuracy, then use the command **round(sqrt(n^1))** to change n to a decimal number, take its square root, and round to the nearest integer. In MATLAB, use the command **digits(50); round(vpa(sqrt(n')))**.

11. Let $p = 123456791$, $q = 987654323$, and $e = 127$. Let the message be $m = 14152019010605$.

1. Compute $m^e \pmod{p}$ and $m^e \pmod{q}$; then use the Chinese remainder theorem to combine these to get $c \equiv m^e \pmod{pq}$.
2. Change one digit of $m^e \pmod{p}$ (for example, this could be caused by some radiation). Now combine this with $m^e \pmod{q}$ to get an incorrect value f for

$m^e \pmod{pq}$. Compute $\gcd(c - f, pq)$. Why does this factor pq ?

The method of (a) for computing $m^e \pmod{pq}$ is attractive since it does not require as large multiprecision arithmetic as working directly mod pq . However, as part (b) shows, if an attacker can cause an occasional bit to fail, then pq can be factored.

12. Suppose that $p = 76543692179$, $q = 343434343453$, and $e = 457$. The ciphertext $c \equiv m^e \pmod{pq}$ is transmitted, but an error occurs during transmission. The received ciphertext is 2304329328016936947195 . The receiver is able to determine that the digits received are correct but that last digit is missing. Determine the missing digit and decrypt the message.

13. Test 38200901201 for primality using the Miller-Rabin test with $a = 2$. Then test using $a = 3$. Note that the first test says that 38200901201 is probably prime, while the second test says that it is composite. A composite number such as 38200901201 that passes the Miller-Rabin test for a number a is called a **strong a -pseudoprime**.

14. There are three users with pairwise relatively prime moduli n_1, n_2, n_3 . Suppose that their encryption exponents are all $e = 3$. The same message m is sent to each of them and you intercept the ciphertexts $c_i \equiv m^3 \pmod{n_i}$ for $i = 1, 2, 3$.

1. Show that $0 \leq m^3 < n_1 n_2 n_3$.
2. Show how to use the Chinese remainder theorem to find m^3 (as an exact integer, *not* only as $m^3 \pmod{n_1 n_2 n_3}$) and therefore m . Do this without factoring.

3. Suppose that

$$n_1 = 2469247531693, \quad n_2 = 11111502225583, \\ n_3 = 44444222221411$$

and the corresponding ciphertexts are

$$359335245251, \quad 10436363975495, \quad 5135984059593.$$

These were all encrypted using $e = 3$. Find the message.

- 15.
1. Choose a 10-digit prime p and a 11-digit prime q . Form $n = pq$.
 2. Let the encryption exponent be $e = 65537$. Write a program that computes the RSA encryptions of all plaintexts m with $1 \leq m < 10^4$. (Do not store or display the results.) The computer probably did this almost instantaneously.

3. Modify your program in (b) so that it computes the encryptions of all m with $1 \leq m < 10^8$ and time how long this takes (if this takes too long, use 10^7 ; if it's too fast to time, use 10^9).
4. Using your timing from (c), estimate how long it will take to encrypt all m with $1 \leq m < n$ (a year is approximately 3×10^7 seconds).

Even this small example shows that it is impractical to make a database of all encryptions in order to attack RSA.

Chapter 10 Discrete Logarithms

10.1 Discrete Logarithms

In the RSA algorithm, we saw how the difficulty of factoring yields useful cryptosystems. There is another number theory problem, namely discrete logarithms, that has similar applications.

Fix a prime p . Let α and β be nonzero integers mod p and suppose

$$\beta \equiv \alpha^x \pmod{p}$$

The problem of finding x is called the **discrete logarithm problem**. If n is the smallest positive integer such that $\alpha^n \equiv 1 \pmod{p}$, we may assume $0 \leq x < n$, and then we denote

$$x = L_\alpha(\beta)$$

and call it the discrete log of β with respect to α (the prime p is omitted from the notation).

For example, let $p = 11$ and let $\alpha = 2$. Since $2^6 \equiv 9 \pmod{11}$, we have $L_2(9) = 6$. Of course, $2^6 \equiv 2^{16} \equiv 2^{26} \equiv 9 \pmod{11}$, so we could consider taking any one of 6, 16, 26 as the discrete logarithm. But we fix the value by taking the smallest nonnegative value, namely 6. Note that we could have defined the discrete logarithm in this case to be the congruence class 6 mod 10. In some ways, this would be more natural, but there are applications where it is convenient to have a number, not just a congruence class.

Often, α is taken to be a primitive root mod p , which means that every β is a power of α ($\text{mod } p$). If α is not a primitive root, then the discrete logarithm will not be defined for certain values of β .

Given a prime p , it is fairly easy to find a primitive root in many cases. See [Exercise 54 in Chapter 3](#).

The discrete log behaves in many ways like the usual logarithm. In particular, if α is a primitive root mod p , then

$$L_\alpha(\beta_1\beta_2) \equiv L_\alpha(\beta_1) + L_\alpha(\beta_2) \pmod{p-1}$$

(see [Exercise 6](#)).

When p is small, it is easy to compute discrete logs by exhaustive search through all possible exponents. However, when p is large this is not feasible. We give some ways of attacking discrete log problems later. However, it is believed that discrete logs are hard to compute in general. This assumption is the basis of several cryptosystems.

The size of the largest primes for which discrete logs can be computed has usually been approximately the same size as the size of largest integers that could be factored (both of these refer to computations that would work for arbitrary numbers of these sizes; special choices of integers will succumb to special techniques, and thus discrete log computations and factorizations work for much larger specially chosen numbers). Compare [Table 10.1](#) with [Table 9.2 in Chapter 9](#).

Table 10.1 Discrete Log Records

Year	Number of Digits of p
2001	120
2005	130
2007	160
2014	180
2016	232

A function $f(x)$ is called a **one-way function** if $f(x)$ is easy to compute, but, given y , it is computationally infeasible to find x with $f(x) = y$. Modular exponentiation is probably an example of such a function. It is easy to compute $\alpha^x \pmod{p}$, but solving $\alpha^x \equiv \beta$ for x is probably hard. Multiplication of large primes can also be regarded as a (probable) one-way function: It is easy to multiply primes but difficult to factor the result to recover the primes. One-way functions have many cryptographic uses.

10.2 Computing Discrete Logs

In this section, we present some methods for computing discrete logarithms. A method based on the birthday attack is discussed in Subsection 12.1.1.

For simplicity, take α to be a primitive root mod p , so $p - 1$ is the smallest positive exponent n such that $\alpha^n \equiv 1 \pmod{p}$. This implies that

$$\alpha^{m_1} \equiv \alpha^{m_2} \pmod{p} \iff m_1 \equiv m_2 \pmod{p-1}.$$

Assume that

$$\beta \equiv \alpha^x, \quad 0 \leq x < p-1.$$

We want to find x .

First, it's easy to determine $x \pmod{2}$. Note that

$$(\alpha^{(p-1)/2})^2 \equiv \alpha^{p-1} \equiv 1 \pmod{p},$$

so $\alpha^{(p-1)/2} \equiv \pm 1 \pmod{p}$ (see Exercise 15 in Chapter 3). However, $p - 1$ is assumed to be the smallest exponent to yield $+1$, so we must have

$$\alpha^{(p-1)/2} \equiv -1 \pmod{p}.$$

Starting with $\beta \equiv \alpha^x \pmod{p}$, raise both sides to the $(p - 1)/2$ power to obtain

$$\beta^{(p-1)/2} \equiv \alpha^{x(p-1)/2} \equiv (-1)^x \pmod{p}.$$

Therefore, if $\beta^{(p-1)/2} \equiv +1$, then x is even; otherwise, x is odd.

Example

Suppose we want to solve $2^x \equiv 9 \pmod{11}$. Since

$$\beta^{(p-1)/2} \equiv 9^5 \equiv 1 \pmod{11},$$

we must have x even. In fact, $x = 6$, as we saw previously.

10.2.1 The Pohlig-Hellman Algorithm

The preceding idea was extended by Pohlig and Hellman to give an algorithm to compute discrete logs when $p - 1$ has only small prime factors. Suppose

$$p - 1 = \prod_i q_i^{r_i}$$

is the factorization of $p - 1$ into primes. Let q^r be one of the factors. We'll compute $L_\alpha(\beta) \pmod{q^r}$. If this can be done for each $q_i^{r_i}$, the answers can be recombined using the Chinese remainder theorem to find the discrete logarithm.

Write

$$x = x_0 + x_1q + x_2q^2 + \cdots \text{ with } 0 \leq x_i \leq q - 1.$$

We'll determine the coefficients x_0, x_1, \dots, x_{r-1} successively, and thus obtain $x \pmod{q^r}$. Note that

$$\begin{aligned} x\left(\frac{p-1}{q}\right) &= x_0\left(\frac{p-1}{q}\right) + (p-1)(x_1 + x_2q + x_3q^2 + \cdots) \\ &= x_0\left(\frac{p-1}{q}\right) + (p-1)n, \end{aligned}$$

where n is an integer. Starting with $\beta \equiv \alpha^x$, raise both sides to the $(p-1)/q$ power to obtain

$$\beta^{(p-1)/q} \equiv \alpha^{x(p-1)/q} \equiv \alpha^{x_0(p-1)/q}(\alpha^{p-1})^n \equiv \alpha^{x_0(p-1)/q} \pmod{p}.$$

The last congruence is a consequence of Fermat's theorem: $\alpha^{p-1} \equiv 1 \pmod{p}$. To find x_0 , simply look at the powers

$$\alpha^{k(p-1)/q} \pmod{p}, \quad k = 0, 1, 2, \dots, q-1,$$

until one of them yields $\beta^{(p-1)/q}$. Then $x_0 = k$. Note that since $\alpha^{m_1} \equiv \alpha^{m_2} \iff m_1 \equiv m_2 \pmod{p-1}$, and since the exponents $k(p-1)/q$ are distinct mod $p-1$, there is a unique k that yields the answer.

An extension of this idea yields the remaining coefficients. Assume that $q^2 | p-1$. Let

$$\beta_1 \equiv \beta \alpha^{-x_0} \equiv \alpha^{q(x_1+x_2q+\dots)} \pmod{p}.$$

Raise both sides to the $(p-1)/q^2$ power to obtain

$$\begin{aligned} \beta_1^{(p-1)/q^2} &\equiv \alpha^{(p-1)(x_1+x_2q+\dots)/q} \\ &\equiv \alpha^{x_1(p-1)/q} (\alpha^{p-1})^{x_2+x_3q+\dots} \\ &\equiv \alpha^{x_1(p-1)/q} \pmod{p}. \end{aligned}$$

The last congruence follows by applying Fermat's theorem. We couldn't calculate $\beta_1^{(p-1)/q^2}$ as $(\beta_1^{p-1})^{1/q^2}$ since fractional exponents cause problems. Note that every exponent we have used is an integer.

To find x_1 , simply look at the powers

$$\alpha^{k(p-1)/q} \pmod{p}, \quad k = 0, 1, 2, \dots, q-1,$$

until one of them yields $\beta_1^{(p-1)/q^2}$. Then $x_1 = k$.

If $q^3 | p-1$, let $\beta_2 \equiv \beta_1 \alpha^{-x_1 q}$ and raise both sides to the $(p-1)/q^3$ power to obtain x_2 . In this way, we can continue until we find that q^{r+1} doesn't divide $p-1$. Since we cannot use fractional exponents, we must stop. But we have determined x_0, x_1, \dots, x_{r-1} , so we know $x \pmod{q^r}$.

Repeat the procedure for all the prime factors of $p-1$. This yields $x \pmod{q_i^{r_i}}$ for all i . The Chinese remainder theorem allows us to combine these into a congruence for $x \pmod{p-1}$. Since $0 \leq x < p-1$, this determines x .

Example

Let $p = 41$, $\alpha = 7$, and $\beta = 12$. We want to solve

$$7^x \equiv 12 \pmod{41}.$$

Note that

$$41 - 1 = 2^3 \cdot 5.$$

First, let $q = 2$ and let's find $x \pmod{2^3}$. Write
 $x \equiv x_0 + 2x_1 + 4x_2 \pmod{8}$.

To start,

$$\beta^{(p-1)/2} \equiv 12^{20} \equiv 40 \equiv -1 \pmod{41},$$

and

$$\alpha^{(p-1)/2} \equiv 7^{20} \equiv -1 \pmod{41}.$$

Since

$$\beta^{(p-1)/2} \equiv (\alpha^{(p-1)/2})^{x_0} \pmod{41},$$

we have $x_0 = 1$. Next,

$$\beta_1 \equiv \beta\alpha^{-x_0} \equiv 12 \cdot 7^{-1} \equiv 31 \pmod{41}.$$

Also,

$$\beta_1^{(p-1)/2^2} \equiv 31^{10} \equiv 1 \pmod{41}.$$

Since

$$\beta_1^{(p-1)/2^2} \equiv (\alpha^{(p-1)/2})^{x_1} \pmod{41},$$

we have $x_1 = 0$. Continuing, we have

$$\beta_2 \equiv \beta_1\alpha^{-2x_1} \equiv 31 \cdot 7^0 \equiv 31 \pmod{41},$$

and

$$\beta_2^{(p-1)/q^3} \equiv 31^5 \equiv -1 \equiv (\alpha^{(p-1)/2})^{x_2} \pmod{41}.$$

Therefore, $x_2 = 1$. We have obtained

$$x \equiv x_0 + 2x_1 + 4x_2 \equiv 1 + 4 \equiv 5 \pmod{8}.$$

Now, let $q = 5$ and let's find $x \pmod{5}$. We have

$$\beta^{(p-1)/5} \equiv 12^8 \equiv 18 \pmod{41}$$

and

$$\alpha^{(p-1)/q} \equiv 7^8 \equiv 37 \pmod{41}.$$

Trying the possible values of k yields

$$37^0 \equiv 1, \quad 37^1 \equiv 37, \quad 37^2 \equiv 16, \quad 37^3 \equiv 18, \quad 37^4 \equiv 10 \pmod{41}.$$

Therefore, 37^3 gives the desired answer, so

$$x \equiv 3 \pmod{5}.$$

Since $x \equiv 5 \pmod{8}$ and $x \equiv 3 \pmod{5}$, we combine these to obtain $x \equiv 13 \pmod{40}$, so $x = 13$. A quick calculation checks that $7^{13} \equiv 12 \pmod{41}$, as desired.

As long as the primes q involved in the preceding algorithm are reasonably small, the calculations can be done quickly. However, when q is large, calculating the numbers $\alpha^{k(p-1)/q}$ for $k = 0, 1, 2, \dots, q - 1$ becomes infeasible, so the algorithm no longer is practical. This means that if we want a discrete logarithm to be hard, we should make sure that $p - 1$ has a large prime factor.

Note that even if $p - 1 = tq$ has a large prime factor q , the algorithm can determine discrete logs mod t if t is composed of small prime factors. For this reason, often β is chosen to be a power of α^t . Then the discrete log is automatically 0 mod t , so the discrete log hides only mod q information, which the algorithm cannot find. If the discrete log x represents a secret (or better, t times a secret), this means that an attacker does not obtain partial information by determining $x \pmod{t}$, since there is no information hidden this way. This idea is used in the Digital Signature Algorithm, which we discuss in [Chapter 13](#).

10.2.2 Baby Step, Giant Step

Eve wants to find x such that $\alpha^x \equiv \beta \pmod{p}$. She does the following. First, she chooses an integer N with $N^2 \geq p - 1$, for example $N = \lceil \sqrt{p-1} \rceil$ (where $\lceil x \rceil$ means round x up to the nearest integer). Then she makes two lists:

1. $\alpha^j \pmod{p}$ for $0 \leq j < N$
2. $\beta\alpha^{-Nk} \pmod{p}$ for $0 \leq k < N$

She looks for a match between the two lists. If she finds one, then

$$\alpha^j \equiv \beta\alpha^{-Nk},$$

so $\alpha^{j+Nk} \equiv \beta$. Therefore, $x = j + Nk$ solves the discrete log problem.

Why should there be a match? Since $0 \leq x < p - 1 \leq N^2$, we can write x in base N as $x = x_0 + Nx_1$ with $0 \leq x_0, x_1 < N$. In fact, $x_1 = [x/N]$ and $x_0 = x - Nx_1$. Therefore,

$$j = x_0, \quad k = x_1$$

gives the desired match.

The list α^j for $j = 0, 1, 2, \dots$ is the set of “Baby Steps” since the elements of the list are obtained by multiplying by α , while the “Giant Steps” are obtained in the second list by multiplying by α^{-N} . It is, of course, not necessary to compute all of the second list. Each element, as it is computed, can be compared with the first list. As soon as a match is found, the computation stops.

The number of steps in this algorithm is proportional to $N \approx \sqrt{p}$ and it requires storing approximately N numbers. Therefore, the method works for primes p up to 10^{20} , or even slightly larger, but is impractical for very large p .

For an example, see [Example 35](#) in the Computer Appendices.

10.2.3 The Index Calculus

The idea is similar to the method of factoring in Subsection [9.4.1](#). Again, we are trying to solve $\beta \equiv \alpha^x \pmod{p}$, where p is a large prime and α is a primitive root.

First, there is a precomputation step. Let B be a bound and let p_1, p_2, \dots, p_m , be the primes less than B . This set of primes is called our **factor base**. Compute $\alpha^k \pmod{p}$ for several values of k . For each such number, try to write it as a product of the primes less than B . If this is not the case, discard α^k . However, if $\alpha^k \equiv \prod p_i^{a_i} \pmod{p}$, then

$$k \equiv \sum a_i L_\alpha(p_i) \pmod{p-1}.$$

When we obtain enough such relations, we can solve for $L_\alpha(p_i)$ for each i .

Now, for random integers r , compute $\beta \alpha^r \pmod{p}$. For each such number, try to write it as a product of primes less than B . If we succeed, we have $\beta \alpha^r \equiv \prod p_i^{b_i} \pmod{p}$, which means

$$L_\alpha(\beta) \equiv -r + \sum b_i L_\alpha(p_i) \pmod{p-1}.$$

This algorithm is effective if p is of moderate size. This means that p should be chosen to have at least 200 digits, maybe more, if the discrete log problem is to be hard.

Example

Let $p = 131$ and $\alpha = 2$. Let $B = 10$, so we are working with the primes $2, 3, 5, 7$. A calculation yields the following:

$$\begin{aligned} 2^1 &\equiv 2 \pmod{131} \\ 2^8 &\equiv 5^3 \pmod{131} \\ 2^{12} &\equiv 5 \cdot 7 \pmod{131} \\ 2^{14} &\equiv 3^2 \pmod{131} \\ 2^{34} &\equiv 3 \cdot 5^2 \pmod{131}. \end{aligned}$$

Therefore,

$$\begin{aligned} 1 &\equiv L_2(2) \pmod{130} \\ 8 &\equiv 3L_2(5) \pmod{130} \\ 12 &\equiv L_2(5) + L_2(7) \pmod{130} \\ 14 &\equiv 2L_2(3) \pmod{130} \\ 34 &\equiv L_2(3) + 2L_2(5) \pmod{130}. \end{aligned}$$

The second congruence yields $L_2(5) \equiv 46 \pmod{130}$. Substituting this into the third congruence yields $L_2(7) \equiv -34 \equiv 96 \pmod{130}$. The fourth congruence yields only the value of $L_2(3) \pmod{65}$ since $\gcd(2, 130) \neq 1$. This gives two choices for $L_2(3) \pmod{130}$. Of course, we could try them and see which works. Or we could use the fifth congruence to obtain $L_2(3) \equiv 72 \pmod{130}$. This finishes the precomputation step.

Suppose now that we want to find $L_2(37)$. Trying a few randomly chosen exponents yields $37 \cdot 2^{43} \equiv 3 \cdot 5 \cdot 7 \pmod{131}$, so

$$L_2(37) \equiv -43 + L_2(3) + L_2(5) + L_2(7) \equiv 41 \pmod{130}.$$

Therefore, $L_2(37) = 41$.

Of course, once the precomputation has been done, it can be reused for computing several discrete logs for the same prime p .

10.2.4 Computing Discrete Logs Mod 4

When $p \equiv 1 \pmod{4}$, the Pohlig-Hellman algorithm computes discrete logs mod 4 quite quickly. What happens when $p \equiv 3 \pmod{4}$? The Pohlig-Hellman algorithm won't work, since it would require us to raise numbers to the $(p - 1)/4$ power, which would yield the ambiguity of a fractional exponent. The surprising fact is that if we have an algorithm that quickly computes discrete logs mod 4 for a prime $p \equiv 3 \pmod{4}$, then we can use it to compute discrete logs mod p quickly. Therefore, it is unlikely that such an algorithm exists.

There is a philosophical reason that we should not expect such an algorithm. A natural point of view is that the discrete log should be regarded as a number mod $p - 1$. Therefore, we should be able to obtain information on the discrete log only modulo the power of 2 that appears in $p - 1$. When $p \equiv 3 \pmod{4}$, this means that asking questions about discrete logs mod 4 is somewhat unnatural. The question is possible only because we normalized the discrete log to be an integer between 0 and $p - 2$. For example, $2^6 \equiv 2^{16} \equiv 9 \pmod{11}$. We defined $L_2(9)$ to be 6 in this case; if we had allowed it also to be 16, we would have two values for $L_2(9)$, namely 6 and 16, that are not congruent mod 4. Therefore, from this point of view, we shouldn't even be asking about $L_2(9) \pmod{4}$.

We need the following lemma, which is similar to the method for computing square roots mod a prime $p \equiv 3 \pmod{4}$ (see Section 3.9).

Lemma

Let $p \equiv 3 \pmod{4}$ be prime, let $r \geq 2$, and let y be an integer. Suppose α and γ are two nonzero numbers mod p such that $\gamma \equiv \alpha^{2^r y} \pmod{p}$. Then

$$\gamma^{(p+1)/4} \equiv \alpha^{2^{r-1}y} \pmod{p}.$$

Proof.

$$\gamma^{(p+1)/4} \equiv \alpha^{(p+1)2^{r-2}y} \equiv \alpha^{2^{r-1}y}(\alpha^{p-1})^{2^{r-2}y} \equiv \alpha^{2^{r-1}y} \pmod{p}.$$

The final congruence is because of Fermat's theorem.

Fix the prime $p \equiv 3 \pmod{4}$ and let α be a primitive root. Assume we have a machine that, given an input β , gives the output $L_\alpha(\beta) \pmod{4}$. As we saw previously, it is easy to compute $L_\alpha(\beta) \pmod{2}$. So the new information supplied by the machine is really only the second bit of the discrete log.

Now assume $\alpha^x \equiv \beta \pmod{p}$. Let $x = x_0 + 2x_1 + 4x_2 + \dots + 2^n x_n$ be the binary expansion of x . Using the $L_\alpha(\beta) \pmod{4}$ machine, we determine x_0 and x_1 . Suppose we have determined x_0, x_1, \dots, x_{r-1} with $r \geq 2$. Let

$$\beta_r \equiv \beta \alpha^{-(x_0 + \dots + 2^{r-1} x_{r-1})} \equiv \alpha^{2^r(x_r + 2x_{r+1} + \dots)}.$$

Using the lemma $r - 1$ times, we find

$$\beta_r^{((p+1)/4)^{r-1}} \equiv \alpha^{2(x_r + 2x_{r+1} + \dots)} \pmod{p}.$$

Applying the $L_\alpha \pmod{4}$ machine to this equation yields the value of x_r . Proceeding inductively, we obtain all the values x_0, x_1, \dots, x_n . This determines x , as desired.

It is possible to make this algorithm more efficient. See, for example, [Stinson1, page 175].

In conclusion, if we believe that finding discrete logs for $p \equiv 3 \pmod{4}$ is hard, then so is computing such discrete logs mod 4.

10.3 Bit Commitment

Alice claims that she has a method to predict the outcome of football games. She wants to sell her method to Bob. Bob asks her to prove her method works by predicting the results of the games that will be played this weekend. “No way,” says Alice. “Then you will simply make your bets and not pay me. If you want me to prove my system works, why don’t I show you my predictions for last week’s games?” Clearly there is a problem here. We’ll show how to resolve it.

Here’s the setup. Alice wants to send a bit b , which is either 0 or 1, to Bob. There are two requirements.

1. Bob cannot determine the value of the bit without Alice’s help.
2. Alice cannot change the bit once she sends it.

One way is for Alice to put the bit in a box, put her lock on it, and send it to Bob. When Bob wants the value of the bit, Alice removes the lock and Bob opens the box. We want to implement this mathematically in such a way that Alice and Bob do not have to be in the same room when the bit is revealed.

Here is a solution. Alice and Bob agree on a large prime $p \equiv 3 \pmod{4}$ and a primitive root α . Alice chooses a random number $x < p - 1$ whose second bit x_1 is b . She sends $\beta \equiv \alpha^x \pmod{p}$ to Bob. We assume that Bob cannot compute discrete logs for p . As pointed out in the last section, this means that he cannot compute discrete logs mod 4. In particular, he cannot determine the value of $b = x_1$. When Bob wants to know the value of b , Alice sends him the full value of x , and by looking at $x \pmod{4}$, he finds b . Alice cannot send a value of x different than the one already used, since Bob checks that

$\beta \equiv \alpha^x \pmod{p}$, and this equation has a unique solution $x < p - 1$.

Back to football: For each game, Alice sends $b = 1$ if she predicts the home team will win, $b = 0$ if she predicts it will lose. After the game has been played, Alice reveals the bit to Bob, who can see whether her predictions were correct. In this way, Bob cannot profit from the information by receiving it before the game, and Alice cannot change her predictions once the game has been played.

Bit commitment can also be accomplished with many other one-way functions. For example, Alice can take a random 100-bit string, followed by the bit b , followed by another 100-bit string. She applies the one-way function to this string and sends the result to Bob. After the game, she sends the full 201-bit string to Bob, who applies the one-way function and compares with what Alice originally sent.

10.4 Diffie-Hellman Key Exchange

An important problem in cryptography is how to establish keys for use in cryptographic protocols such as DES or AES, especially when the two parties are widely separated. Public key methods such as RSA provide one solution. In the present section, we describe a different method, due to Diffie and Hellman, whose security is very closely related to the difficulty of computing discrete logarithms.

There are several technical implementation issues related to any key distribution scheme. Some of these are discussed in [Chapter 15](#). In the present section, we restrict ourselves to the basic Diffie-Hellman algorithm. For more discussion of some security concerns about implementations of the Diffie-Hellman protocol, see [Adrian et al.].

Here is how Alice and Bob establish a private key K . All of their communications in the following algorithm are over public channels.

1. Either Alice or Bob selects a large prime number p for which the discrete logarithm problem is hard and a primitive root $\alpha \pmod{p}$. Both p and α can be made public.
2. Alice chooses a secret random x with $1 \leq x \leq p - 2$, and Bob selects a secret random y with $1 \leq y \leq p - 2$.
3. Alice sends $\alpha^x \pmod{p}$ to Bob, and Bob sends $\alpha^y \pmod{p}$ to Alice.
4. Using the messages that they each have received, they can each calculate the session key K . Alice calculates K by $K \equiv (\alpha^y)^x \pmod{p}$, and Bob calculates K by $K \equiv (\alpha^x)^y \pmod{p}$.

There is no reason that Alice and Bob need to use all of K as their key for their communications. Now that they have the same number K , they can use some prearranged procedure to produce a key. For example, they could use the middle 56 bits of K to obtain a DES key.

Suppose Eve listens to all the communications between Alice and Bob. She will know α^x and α^y . If she can compute discrete logs, then she can find the discrete log of α^x to obtain x . Then she raises α^y to the power x to obtain $\alpha^{xy} \equiv K$. Once Eve has K , she can use the same procedure as Alice and Bob to extract a communication key. Therefore, if Eve can compute discrete logs, she can break the system.

However, Eve does not necessarily need to compute x or y to find K . What she needs to do is solve the following:

Computational Diffie-Hellman Problem: Let p be prime and let α be a primitive root mod p . Given $\alpha^x \pmod{p}$ and $\alpha^y \pmod{p}$, find $\alpha^{xy} \pmod{p}$.

It is not known whether or not this problem is easier than computing discrete logs. The reasoning above shows that it is no harder than computing discrete logs. A related problem is the following:

Decision Diffie-Hellman Problem: Let p be prime and let α be a primitive root mod p . Given $\alpha^x \pmod{p}$ and $\alpha^y \pmod{p}$, and $c \not\equiv 0 \pmod{p}$, decide whether or not $c \equiv \alpha^{xy} \pmod{p}$.

In other words, if Eve claims that she has found c with $c \equiv \alpha^{xy} \pmod{p}$, and offers to sell you this information, can you decide whether or not she is telling the truth? Of course, if you can solve the computational Diffie-Hellman problem, then you simply compute $\alpha^{xy} \pmod{p}$ and check whether it is c (and then you can ignore Eve's offer).

Conversely, does a method for solving the decision Diffie-Hellman problem yield a solution to the computational Diffie-Hellman problem? This is not known at present. One obvious method is to choose many values of c and check each value until one equals $\alpha^{xy} \pmod{p}$. But this brute force method takes at least as long as computing discrete logarithms by brute force, so is impractical. There are situations involving elliptic curves, analogous to the present setup, where a fast solution is known for the decision Diffie-Hellman problem but no practical solution is known for the computational Diffie-Hellman problem (see [Exercise 8](#) in [Chapter 22](#)).

10.5 The ElGamal Public Key Cryptosystem

In Chapter 9, we studied a public key cryptosystem whose security is based on the difficulty of factoring. It is also possible to design a system whose security relies on the difficulty of computing discrete logarithms. This was done by ElGamal in 1985. This system does not quite fit the definition of a public key cryptosystem given at the end of Chapter 9, since the set of possible plaintexts (integers mod p) is not the same as the set of possible ciphertexts (pairs of integers (r, t) mod p). However, this technical point will not concern us.

Alice wants to send a message m to Bob. Bob chooses a large prime p and a primitive root α . Assume m is an integer with $0 \leq m < p$. If m is larger, break it into smaller blocks. Bob also chooses a secret integer b and computes $\beta \equiv \alpha^b \pmod{p}$. The information (p, α, β) is made public and is Bob's public key. Alice does the following:

1. Downloads (p, α, β)
2. Chooses a secret random integer k and computes $r \equiv \alpha^k \pmod{p}$
3. Computes $t \equiv \beta^k m \pmod{p}$
4. Sends the pair (r, t) to Bob

Bob decrypts by computing

$$tr^{-b} \equiv m \pmod{p}.$$

This works because

$$tr^{-b} \equiv \beta^k m (\alpha^k)^{-b} \equiv (\alpha^b)^k m \alpha^{-bk} \equiv m \pmod{p}.$$

If Eve determines b , then she can also decrypt by the same procedure that Bob uses. Therefore, it is important for Bob to keep b secret. The numbers α and β are public, and $\beta \equiv \alpha^b \pmod{p}$. The difficulty of computing discrete logs is what keeps b secure.

Since k is a random integer, β^k is a random nonzero integer mod p . Therefore, $t \equiv \beta^k m \pmod{p}$ is m multiplied by a random integer, and t is random mod p (unless $m = 0$, which should be avoided, of course). Therefore, t gives Eve no information about m . Knowing r does not seem to give Eve enough additional information.

The integer k is difficult to determine from r , since this is again a discrete logarithm problem. However, if Eve finds k , she can then calculate $t\beta^{-k}$, which is m .

It is important that a different random k be used for each message. Suppose Alice encrypts messages m_1 and m_2 for Bob and uses the same value k for each message. Then r will be the same for both messages, so the ciphertexts will be (r, t_1) and (r, t_2) . If Eve finds out the plaintext m_1 , she can also determine m_2 , as follows. Note that

$$t_1/m_1 \equiv \beta^k \equiv t_2/m_2 \pmod{p}.$$

Since Eve knows t_1 and t_2 , she computes $m_2 \equiv t_2 m_1 / t_1 \pmod{p}$.

In [Chapter 21](#), we'll meet an analog of the ElGamal method that uses elliptic curves.

10.5.1 Security of ElGamal Ciphertexts

Suppose Eve claims to have obtained the plaintext m corresponding to an RSA ciphertext c . It is easy to verify her claim: Compute $m^e \pmod{n}$ and check whether this equals c . Now suppose instead that Eve claims to possess the message m corresponding to an ElGamal encryption (r, t) . Can you verify her claim? It turns out that this is as hard as the decision Diffie-Hellman problem from [Section 10.4](#). In this aspect, the ElGamal algorithm is therefore much different than the RSA algorithm (of course, if some randomness is added to an RSA plaintext through OAEP, for example, then RSA encryption has a similar property).

Proposition

A machine that solves Decision Diffie-Hellman problems mod p can be used to decide the validity of mod p ElGamal ciphertexts, and a machine that decides the validity of mod p ElGamal ciphertexts can be used to solve Decision Diffie-Hellman problems mod p .

Proof. Suppose first that you have a machine M_1 that can decide whether an ElGamal decryption is correct. In other words, when given the inputs $p, \alpha, \beta, (r, t), m$, the machine outputs “yes” if m is the decryption of (r, t) and outputs “no” otherwise. Let’s use this machine to solve the decision Diffie-Hellman problem. Suppose you are given α^x and α^y , and you want to decide whether or not $c \equiv \alpha^{xy} \pmod{p}$. Let $\beta = \alpha^x$ and $r = \alpha^y \pmod{p}$.

Moreover, let $t = c$ and $m = 1$. Input

$$p, \quad \alpha, \quad \beta, \quad (\alpha^x, \alpha^y), \quad 1$$

into M_1 . Note that in the present setup, x is the secret integer b and α^y takes the place of the $r \equiv \alpha^k$. The correct decryption of $(r, t) = (\alpha^y, \alpha^{xy})$ is $tr^{-b} \equiv cr^{-x} \equiv c\alpha^{-xy} \pmod{p}$. Therefore, M_1 outputs “yes” exactly when $m = 1$ is the same as $c\alpha^{-xy} \pmod{p}$,

namely when $c \equiv \alpha^{xy} \pmod{p}$. This solves the decision Diffie-Hellman problem.

Conversely, suppose you have a machine M_2 that can solve the decision Diffie-Hellman problem. This means that if you give M_2 inputs $p, \alpha, \alpha^x, \alpha^y, c$, then M_2 outputs “yes” if $c \equiv \alpha^{xy}$ and outputs “no” if not. Let m be the claimed decryption of the ElGamal ciphertext (r, t) . Input $\beta \equiv \alpha^b$ as α^x , so $x = b$, and input $r \equiv \alpha^k$ as α^y so $y = k$. Input $tm^{-1} \pmod{p}$ as c . Note that m is the correct plaintext for the ciphertext (r, t) if and only if $m \equiv tr^{-a} \equiv t\alpha^{-xy}$, which happens if and only if $tm^{-1} \equiv \alpha^{xy}$. Therefore, m is the correct plaintext if and only if $c \equiv tm^{-1}$ is the solution to the Diffie-Hellman problem. Therefore, with these inputs, M_2 outputs “yes” exactly when m is the correct plaintext.

The reasoning just used can also be used to show that solving the computational Diffie-Hellman problem is equivalent to breaking the ElGamal system:

Proposition

A machine that solves computational Diffie-Hellman problems mod p can be used to decrypt mod p ElGamal ciphertexts, and a machine that decrypts mod p ElGamal ciphertexts can be used to solve computational Diffie-Hellman problems mod p .

Proof. If we have a machine M_3 that can decrypt all ElGamal ciphertexts, then input $\beta \equiv \alpha^x$ (so $a = x$) and $r \equiv \alpha^y$. Take any nonzero value for t . Then M_3 outputs $m \equiv tr^{-a} \equiv t\alpha^{-xy}$. Therefore, $tm^{-1} \pmod{p}$ yields the solution α^{xy} to the computational Diffie-Hellman problem.

Conversely, suppose we have a machine M_4 that can solve computational Diffie-Hellman problems. If we have

an ElGamal ciphertext (r, t) , then we input
 $\alpha^x = \alpha^a \equiv \beta$ and $\alpha^y \equiv \alpha^k \equiv r$. Then M_4 outputs
 $\alpha^{xy} \equiv \alpha^{ak}$. Since $m \equiv tr^{-a} \equiv t\alpha^{-ak}$, we obtain the
plaintext m .

10.6 Exercises

1.
 1. Let $p = 13$. Compute $L_2(3)$.
 2. Show that $L_2(11) = 7$.
2.
 1. Let $p = 17$. Compute $L_3(2)$.
 2. Show that $L_3(15) = 6$.
3.
 1. Compute $6^5 \pmod{11}$.
 2. Let $p = 11$. Then 2 is a primitive root. Suppose $2^x \equiv 6 \pmod{11}$. Without finding the value of x , determine whether x is even or odd.
4. Let $p = 19$. Then 2 is a primitive root. Use the Pohlig-Hellman method to compute $L_2(14)$.
5. It can be shown that 5 is a primitive root for the prime 1223. You want to solve the discrete logarithm problem $5^x \equiv 3 \pmod{1223}$. Given that $3^{611} \equiv 1 \pmod{1223}$, determine whether x is even or odd.
6.
 1. Let α be a primitive root mod p . Show that
$$L_\alpha(\beta_1\beta_2) \equiv L_\alpha(\beta_1) + L_\alpha(\beta_2) \pmod{p-1}.$$
(Hint: You need the proposition in [Section 3.7](#).)
 2. More generally, let α be arbitrary. Show that
$$L_\alpha(\beta_1\beta_2) \equiv L_\alpha(\beta_1) + L_\alpha(\beta_2) \pmod{\text{ord}_p(\alpha)},$$
where $\text{ord}_p(\alpha)$ is defined in [Exercise 53](#) in [Chapter 3](#).
7. Let $p = 101$, so 2 is a primitive root. It can be shown that $L_2(3) = 69$ and $L_2(5) = 24$.
 1. Using the fact that $24 = 2^3 \cdot 3$, evaluate $L_2(24)$.
 2. Using the fact that $5^3 \equiv 24 \pmod{101}$, evaluate $L_2(24)$.
8. The number 12347 is prime. Suppose Eve discovers that $2^{10000} \cdot 79 \equiv 2^{5431} \pmod{12347}$. Find an integer k with $0 < k < 12347$ such that $2^k \equiv 79 \pmod{12347}$.

9. Suppose you know that

$$3^6 \equiv 44 \pmod{137}, \quad 3^{10} \equiv 2 \pmod{137}.$$

Find a value of x with $0 \leq x \leq 135$ such that

$$3^x \equiv 11 \pmod{137}.$$

10. Let p be a large prime and suppose $\alpha^{10^{18}} \equiv 1 \pmod{p}$. Suppose $\beta \equiv \alpha^k \pmod{p}$ for some integer k .

1. Explain why we may assume that $0 \leq k < 10^{18}$.

2. Describe a BabyStep, Giant Step method to find k . (Hint:
One list can contain numbers of the form $\beta\alpha^{-10^9 j}$.)

11. 1. Suppose you have a random 500-digit prime p . Suppose some people want to store passwords, written as numbers. If x is the password, then the number $2^x \pmod{p}$ is stored in a file. When y is given as a password, the number $2^y \pmod{p}$ is compared with the entry for the user in the file. Suppose someone gains access to the file. Why is it hard to deduce the passwords?

2. Suppose p is instead chosen to be a five-digit prime. Why would the system in part (a) not be secure?

12. Let's reconsider [Exercise 55 in Chapter 3](#) from the point of view of the Pohlig-Hellman algorithm. The only prime q is 2. For k as in that exercise, write $k = x_0 + 2x_1 + \dots + 2^{15}x_{15}$.

1. Show that the Pohlig-Hellman algorithm yields

$$x_0 = x_1 = \dots = x_{10} = 0$$

and

$$2 = \beta = \beta_1 = \dots = \beta_{11}.$$

2. Use the Pohlig-Hellman algorithm to compute k .

13. In the Diffie-Hellman Key Exchange protocol, suppose the prime is $p = 17$ and the primitive root is $\alpha = 3$. Alice's secret is $a = 3$ and Bob's secret is $b = 5$. Describe what Alice and Bob send each other and determine the shared secret that they obtain.

14. In the Diffie-Hellman Key Exchange protocol, Alice thinks she can trick Eve by choosing her secret to be $a = 0$. How will Eve recognize that Alice made this choice?

15. In the Diffie-Hellman key exchange protocol, Alice and Bob choose a primitive root α for a large prime p . Alice sends $x_1 \equiv \alpha^a \pmod{p}$ to Bob, and Bob sends $x_2 \equiv \alpha^b \pmod{p}$ to Alice. Suppose Eve bribes Bob to tell her the values of b and x_2 .

However, he neglects to tell her the value of α . Suppose $\gcd(b, p - 1) = 1$. Show how Eve can determine α from the knowledge of p , x_2 , and b .

16. In the ElGamal cryptosystem, Alice and Bob use $p = 17$ and $\alpha = 3$. Bob chooses his secret to be $a = 6$, so $\beta = 15$. Alice sends the ciphertext $(r, t) = (7, 6)$. Determine the plaintext m .
17. Consider the following Baby Step, Giant Step attack on RSA, with public modulus n . Eve knows a plaintext m and a ciphertext c with $\gcd(c, n) = 1$. She chooses $N^2 \geq n$ and makes two lists: The first is $c^j \pmod{n}$ for $0 \leq j < N$. The second is $mc^{-Nk} \pmod{n}$ for $0 \leq k < N$.
 1. Why is there always a match between the two lists, and how does a match allow Eve to find the decryption exponent d ?
 2. Your answer to (a) is probably partly false. What you have really found is an exponent d such that $c^d \equiv m \pmod{n}$. Give an example of a plaintext-ciphertext pair where the d you find is not the encryption exponent. (However, usually d is very close to being the correct decryption exponent.)
 3. Why is this not a useful attack on RSA? (Hint: How long are the lists compared to the time needed to factor n by trial division?)
18. Alice and Bob are using the ElGamal public key cryptosystem, but have set it up so that only Alice, Bob, and a few close associates (not Eve) know Bob's public key. Suppose Alice is sending the message *dismiss Eve* to Bob, but Eve intercepts the message and prevents Bob from receiving it. How can Eve change the message to *promote Eve* before sending it to Bob?

10.7 Computer Problems

1. Let $p = 53047$. Verify that $L_3(8576) = 1234$.
2. Let $p = 31$. Evaluate $L_3(24)$.
3. Let $p = 3989$. Then 2 is a primitive root mod p .
 1. Show that $L_2(3925) = 2000$ and $L_2(1046) = 3000$.
 2. Compute $L_2(3925 \cdot 1046)$. (Note: The answer should be less than 3988.)
4. Let $p = 1201$.
 1. Show that $11^{1200/q} \not\equiv 1 \pmod{1201}$ for $q = 2, 3, 5$.
 2. Use method of [Exercise 54 in Chapter 3](#) plus the result of part (a) to show that 11 is a primitive root mod 1201.
 3. Use the Pohlig-Hellman algorithm to find $L_{11}(2)$.
 4. Use the Baby Step, Giant Step method to find $L_{11}(2)$.

Chapter 11 Hash Functions

11.1 Hash Functions

A basic component of many cryptographic algorithms is what is known as a hash function. When a hash function satisfies certain non-invertibility properties, it can be used to make many algorithms more efficient. In the following, we discuss the basic properties of hash functions and attacks on them. We also briefly discuss the random oracle model, which is a method of analyzing the security of algorithms that use hash functions. Later, in [Chapter 13](#), hash functions will be used in digital signature algorithms. They also play a role in security protocols in [Chapter 15](#), and in several other situations.

A **cryptographic hash function** h takes as input a message of arbitrary length and produces as output a **message digest** of fixed length, for example, 256 bits as depicted in [Figure 11.1](#). Certain properties should be satisfied:

1. Given a message m , the message digest $h(m)$ can be calculated very quickly.
2. Given a y , it is computationally infeasible to find an m' with $h(m') = y$ (in other words, h is a **one-way**, or **preimage resistant**, function). Note that if y is the message digest of some message, we are not trying to find this message. We are only looking for some m' with $h(m') = y$.
3. It is computationally infeasible to find messages m_1 and m_2 with $h(m_1) = h(m_2)$ (in this case, the function h is said to be **strongly collision resistant**).

Figure 11.1 A Hash Function

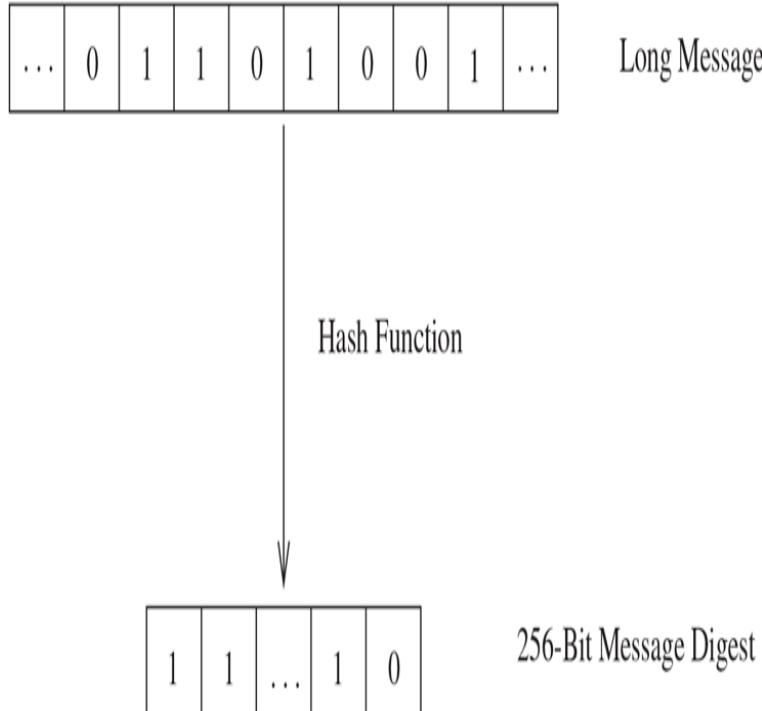


Figure 11.1 Full Alternative Text

Note that since the set of possible messages is much larger than the set of possible message digests, there should always be many examples of messages m_1 and m_2 with $h(m_1) = h(m_2)$. The requirement (3) says that it should be hard to find examples. In particular, if Bob produces a message m and its hash $h(m)$, Alice wants to be reasonably certain that Bob does not know another message m' with $h(m') = h(m)$, even if both m and m' are allowed to be random strings of symbols.

Preimage resistance and collision resistance are closely related, but we list them separately because they are used in slightly different circumstances. The following argument shows that, for our hash functions, collision resistance implies preimage resistance: Suppose H is not preimage resistant. Take a random x and compute $y = H(x)$. If H is not preimage resistant, we can quickly find x' with $H(x') = y = H(x)$. Because H is many-to-one, it is likely that $x \neq x'$, so we have a collision, contradicting the collision resistance of H . However, there are examples that show that for arbitrary

functions, collision resistance does not imply preimage resistance. See [Exercise 12](#).

In practice, it is sometimes sufficient to weaken (3) to require H to be **weakly collision resistant**. This means that given x , it is computationally infeasible to find $x' \neq x$ with $H(x') = H(x)$. This property is also called **second preimage resistance**.

Requirement (3) is the hardest one to satisfy. In fact, in 2004, Wang, Feng, Lai, and Yu (see [Wang et al.]) found many examples of collisions for the popular hash functions MD4, MD5, HAVAL-128, and RIPEMD. The MD5 collisions have been used by Ondrej Mikle [Mikle] to create two different and meaningful documents with the same hash, and the paper [Lenstra et al.] shows how to produce examples of X.509 certificates (see [Section 15.5](#)) with the same MD5 hash (see also [Exercise 15](#)). This means that a valid digital signature (see [Chapter 13](#)) on one certificate is also valid for the other certificate, hence it is impossible for someone to determine which is the certificate that was legitimately signed by a Certification Authority. It has been reported that weaknesses in MD5 were part of the design of the Flame malware, which attacked several computers in the Middle East, including Iran's oil industry, from 2010 to 2012.

In 2005, Wang, Yin, and Yu [Wang et al. 2] predicted that collisions could be found for the hash function SHA-1 with around 2^{69} calculations, which is much better than the expected 2^{80} calculations required by the birthday attack (see [Section 12.1](#)). In addition, they found collisions in a smaller 60-round version of SHA-1. These weaknesses were a cause for concern for using these hash algorithms and led to research into replacements. Finally, in 2017, a joint project between CWI Amsterdam and Google Research found collisions for SHA-1 [Stevens

et al.]. Although SHA-1 is still common, it is starting to be used less and less.

One of the main uses of hash functions is in digital signatures. Since the length of a digital signature is often at least as long as the document being signed, it is much more efficient to sign the hash of a document rather than the full document. This will be discussed in [Chapter 13](#).

Hash functions may also be employed as a check on data integrity. The question of data integrity comes up in basically two scenarios. The first is when the data (encrypted or not) are being transmitted to another person and a noisy communication channel introduces errors to the data. The second occurs when an observer rearranges the transmission in some manner before it gets to the receiver. Either way, the data have become corrupted.

For example, suppose Alice sends Bob long messages about financial transactions with Eve and encrypts them in blocks. Perhaps Eve deduces that the tenth block of each message lists the amount of money that is to be deposited to Eve's account. She could easily substitute the tenth block from one message into another and increase the deposit.

In another situation, Alice might send Bob a message consisting of several blocks of data, but one of the blocks is lost during transmission. Bob might never realize that the block is missing.

Here is how hash functions can be used. Say we send $(m, h(m))$ over the communications channel and it is received as (M, H) . To check whether errors might have occurred, the recipient computes $h(M)$ and sees whether it equals H . If any errors occurred, it is likely that $h(M) \neq H$, because of the collision-resistance properties of h .

Example

Let n be a large integer. Let $h(m) = m \pmod n$ be regarded as an integer between 0 and $n - 1$. This function clearly satisfies (1). However, (2) and (3) fail: Given y , let $m = y$. Then $h(m) = y$. So h is not one-way. Similarly, choose any two values m_1 and m_2 that are congruent mod n . Then $h(m_1) = h(m_2)$, so h is not strongly collision resistant.

Example

The following example, sometimes called the discrete log hash function, is due to Chaum, van Heijst, and Pfitzmann [Chaum et al.]. It satisfies (2) and (3) but is much too slow to be used in practice. However, it demonstrates the basic idea of a hash function.

First we select a large prime number p such that $q = (p - 1)/2$ is also prime (see [Exercise 15](#) in [Chapter 13](#)). We now choose two primitive roots α_1 and α_2 for p . Since α_1 is a primitive root, there exists a such that $\alpha_1^a \equiv \alpha_2 \pmod p$. However, we assume that a is not known (finding a , if not given it in advance, involves solving a discrete log problem, which we assume is hard).

The hash function h will map integers mod q^2 to integers mod p . Therefore, the message digest usually contains approximately half as many bits as the message. This is not as drastic a reduction in size as is usually required in practice, but it suffices for our purposes.

Write $m = x_0 + x_1q$ with $0 \leq x_0, x_1 \leq q - 1$. Then define

$$h(m) \equiv \alpha_1^{x_0} \alpha_2^{x_1} \pmod p.$$

The following shows that the function h is probably strongly collision resistant.

Proposition

If we know messages $m \neq m'$ with $h(m) = h(m')$, then we can determine the discrete logarithm $a = L_{\alpha_1}(\alpha_2)$.

Proof

Write $m = x_0 + x_1 q$ and $m' = x'_0 + x'_1 q$. Suppose

$$\alpha_1^{x_0} \alpha_2^{x_1} \equiv \alpha_1^{x'_0} \alpha_2^{x'_1} \pmod{p}.$$

Using the fact that $\alpha_2 \equiv \alpha_1^a \pmod{p}$, we rewrite this as

$$\alpha_1^{a(x_1 - x'_1) - (x'_0 - x_0)} \equiv 1 \pmod{p}.$$

Since α_1 is a primitive root mod p , we know that

$\alpha_1^k \equiv 1 \pmod{p}$ if and only if $k \equiv 0 \pmod{p-1}$. In our case, this means that

$$a(x_1 - x'_1) - (x'_0 - x_0) \equiv 0 \pmod{p-1}.$$

Let $d = \gcd(x_1 - x'_1, p-1)$. There are exactly d solutions to the preceding congruence (see Subsection 3.3.1), and they can be found quickly. By the choice of p , the only factors of $p-1$ are 1, 2, q , $p-1$. Since $0 \leq x_1, x'_1 \leq q-1$, it follows that $-(q-1) \leq x_1 - x'_1 \leq q-1$. Therefore, if $x_1 - x'_1 \neq 0$, then it is a nonzero multiple of d of absolute value less than q . This means that $d \neq q, p-1$, so $d = 1$ or 2 . Therefore, there are at most two possibilities for a . Calculate α_1^a for each possibility; only one of them will yield α_2 . Therefore, we obtain a , as desired.

On the other hand, if $x_1 - x'_1 = 0$, then the preceding yields $x'_0 - x_0 \equiv 0 \pmod{p-1}$. Since $-(q-1) \leq x'_0 - x_0 \leq q-1$, we must have $x'_0 = x_0$. Therefore, $m = m'$, contrary to our assumption.

It is now easy to show that h is preimage resistant.

Suppose we have an algorithm g that starts with a message digest y and quickly finds an m with $h(m) = y$. In this case, it is easy to find $m_1 \neq m_2$ with $h(m_1) = h(m_2)$: Choose a random m and compute $y = h(m)$, then compute $g(y)$. Since h maps q^2 messages to $p - 1 = 2q$ message digests, there are many messages m' with $h(m') = h(m)$. It is therefore not very likely that $m' = m$. If it is, try another random m . Soon, we should find a collision, that is, messages $m_1 \neq m_2$ with $h(m_1) = h(m_2)$. The preceding proposition shows that we can then solve a discrete log problem. Therefore, it is unlikely that such an algorithm g exists.

As we mentioned earlier, this hash function is good for illustrative purposes but is impractical because of its slow nature. Although it can be computed efficiently via repeated squaring, it turns out that even repeated squaring is too slow for practical applications. In applications such as electronic commerce, the extra time required to perform the multiplications in software is prohibitive.

11.2 Simple Hash Examples

There are many families of hash functions. The discrete log hash function that we described in the previous section is too slow to be of practical use. One reason is that it employs modular exponentiation, which makes its computational requirements about the same as RSA or ElGamal. Even though modular exponentiation is fast, it is not fast enough for the massive inputs that are used in some situations. The hash functions described in this section and the next are easily seen to involve only very basic operations on bits and therefore can be carried out much faster than procedures such as modular exponentiation.

We now describe the basic idea behind many cryptographic hash functions by giving a simple hash function that shares many of the basic properties of hash functions that are used in practice. This hash function is not an industrial-strength hash function and should never be used in any system.

Suppose we start with a message m of arbitrary length L . We may break m into n -bit blocks, where n is much smaller than L . We denote these n -bit blocks by m_j , and thus represent $m = [m_1, m_2, \dots, m_l]$. Here $l = \lceil L/n \rceil$, and the last block m_l is padded with zeros to ensure that it has n bits.

We write the j th block m_j as a row vector

$$m_j = [m_{j1}, m_{j2}, m_{j3}, \dots, m_{jn}],$$

where each m_{ji} is a bit.

Now, we may stack these row vectors to form an array. Our hash $h(m)$ will have n bits, where we calculate the i

th bit as the XOR along the i th column of the matrix, that is $h_i = m_{1i} \oplus m_{2i} \oplus \dots \oplus m_{li}$. We may visualize this as

$$\begin{array}{cccc}
 m_{11} & m_{12} & \cdots & m_{1n} \\
 m_{21} & m_{22} & \cdots & m_{2n} \\
 \vdots & \vdots & \ddots & \vdots \\
 m_{l1} & m_{l2} & \cdots & m_{ln} \\
 \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
 \oplus & \oplus & \oplus & \oplus \\
 \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
 [c_1 & c_2 & \cdots & c_n] & = & h(m).
 \end{array}$$

This hash function is able to take an arbitrary length message and output an n -bit message digest. It is not considered cryptographically secure, though, since it is easy to find two messages that hash to the same value ([Exercise 9](#)).

Practical cryptographic hash functions typically make use of several other bit-level operations in order to make it more difficult to find collisions. [Section 11.4](#) contains many examples of such operations.

One operation that is often used is bit rotation. We define the right rotation operation

$$R^y(m)$$

as the result of shifting m to the right by y positions and wrapping the rightmost y bits around, placing them in leftmost y bit locations. Then $R^{-y}(m)$ gives a similar rotation of m by y places to the left.

We may modify our simple hash function above by requiring that block m_j is left rotated by $j - 1$, to produce a new block $m'_j = R^{-(j-1)}(m_j)$. We may now arrange the m'_j in columns and define a new, simple hash function by XORing these columns. Thus, we get

$$\begin{array}{ccccccc}
m_{11} & m_{12} & \cdots & m_{1n} \\
m_{22} & m_{23} & \cdots & m_{21} \\
m_{33} & m_{34} & \cdots & m_{32} \\
\vdots & \vdots & \ddots & \vdots \\
m_{ll} & m_{l,l+1} & \cdots & m_{l,l-1} \\
\downarrow & \downarrow & \downarrow & \downarrow \\
\oplus & \oplus & \oplus & \oplus \\
\downarrow & \downarrow & \downarrow & \downarrow \\
[c_1 & c_2 & \cdots & c_n] & = & h(m).
\end{array}$$

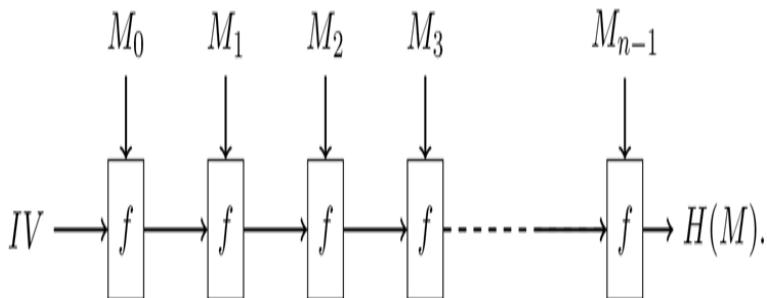
This new hash function involving rotations mixes the bits in one position with those in another, but it is still easy to find collisions ([Exercise 9](#)). Building a cryptographic hash requires considerably more tricks than just rotating. In later sections, we describe hash functions that are used in practice. They use the techniques of the present section, coupled with many more ways of mixing the bits.

11.3 The Merkle-Damgård Construction

Until recently, most hash functions used a form of the Merkle-Damgård construction. It was invented independently by Ralph Merkle in 1979 and Ivan Damgård in 1989. The main ingredient is a function f , usually called a **compression function**. It takes two bitstrings as inputs, call them H and M , and outputs a bitstring $H' = f(H, M)$ of the same length as H . For example, M could have length 512 and H could have length 256. These are the sizes that the hash function SHA-256 uses, and we'll use them for concreteness. The message that is to be hashed is suitably padded so that its length is a multiple of 512, and then broken into n blocks of length 512:

$$M_0 || M_1 || M_2 || \cdots || M_{n-1}.$$

An initial value IV is set. Then the blocks are fed one-by-one into f and the final output is the hash value:



11.3-1 Full Alternative Text

This construction is very natural: The blocks are read from the message one at a time and stirred into the mix with the previous blocks. The final result is the hash value.

Over the years, some disadvantages of the method have been discovered. One is called the **length extension attack**. For example, suppose Alice wants to ensure that her message M to Bob has not been tampered with. They both have a secret key K , so Alice prepends M with K to get $K||M$. She sends both M and $H(K||M)$ to Bob. Since Bob also knows K , he computes $H(K||M)$ and checks that it agrees with the hash value sent by Alice. If so, Bob concludes that the message is authentic.

Since Eve does not know K , she cannot send her own message M' along with $H(K||M')$. But, because of the iterative form of the hash function, Eve can append blocks M'' to M if she intercepts Alice's communication. Then Eve sends

$$M||M'' \text{ and } H(K||M||M'') = f(H(K||M), M'')$$

to Bob. Since she knows $H(K||M)$, she can produce a message that Bob will regard as authentic. Of course, this attack can be thwarted by using $M||K$ instead of $K||M$, but it points to a weakness in the construction.

However, using $H(M||K)$ might also cause problems if Eve discovers a way of producing collisions for H . Namely, Eve finds M_1 and M_2 with $H(M_1) = H(M_2)$. If Eve can arrange this so that M_1 is a good message and M_2 is a bad message (see [Section 12.1](#)), then Eve arranges for Alice to authenticate M_1 by computing $H(M_1||K)$, which equals $H(M_2||K)$. This means that Alice has also authenticated M_2 .

Another attack was given by Daum and Luks in 2005. Suppose Alice is using a high-level document language such as PostScript, which is really a program rather than just a text file. The file begins with a preamble that identifies the file as PostScript and gives some instructions. Then the content of the file follows.

Suppose Eve is able to find random strings R_1 and R_2 such that

$$H(\text{preamble}; \text{put}(R_1)) = H(\text{preamble}; \text{put}(R_2)),$$

where $\text{put}(R_i)$ instructs the PostScript program to put the string R_i in a certain register. In other words, we are assuming that Eve has found a collision of this form. If any string S is appended to these messages, there is still a collision

$$H(\text{preamble}; \text{put}(R_1) || S) = H(\text{preamble}; \text{put}(R_2) || S)$$

because of the iterative nature of the hash algorithm (we are ignoring the effects of padding).

Of course, Eve has an evil document T_1 , perhaps saying that Alice (who is a bank president) gives Eve access to the bank's vault. Eve also produces a document T_2 that Alice will be willing to sign, for example, a petition to give bank presidents tax breaks. Eve then produces two messages:

$$\begin{aligned} Y_1 &= \text{preamble}; \text{put}(R_1); \text{put}(R_1); \text{if } (=) \text{ then } T_1 \text{ else } T_2 \\ Y_2 &= \text{preamble}; \text{put}(R_2); \text{put}(R_1); \text{if } (=) \text{ then } T_1 \text{ else } T_2. \end{aligned}$$

For example, Y_2 puts R_1 into a stack, then puts in R_2 . They are not equal, so T_2 is produced. Eve now has two Postscript files, Y_1 and Y_2 , with $H(Y_1) = H(Y_2)$. As we'll see in [Chapter 13](#), it is standard for Alice to sign the hash of a message rather than the message itself. Eve shows Y_2 to Alice, who compiles it. The output is the petition that Alice is happy to sign. So Alice signs $H(Y_2)$. But this means Alice has also signed $H(Y_1)$. Eve takes Y_1 to the bank, along with Alice's signature on its hash value. The security officer at the bank checks that the signature is valid, then opens the document, which says that Alice grants Eve access to the bank's vault. This potentially costly forgery relies on Eve being able to find a collision, but again it shows a weakness in the construction if there is a possibility of finding collisions.

11.4 SHA-2

In this section and the next, we look at what is involved in making a real cryptographic hash function. Unlike block ciphers, where there are many block ciphers to choose from, there are only a few hash functions that are used in practice. The most notable of these are the Secure Hash Algorithm (SHA) family, the Message Digest (MD) family, and the RIPEMD-160 message digest algorithm. The original MD algorithm was never published, and the first MD algorithm to be published was MD2, followed by MD4 and MD5. Weaknesses in MD2 and MD4 were found, and MD5 was proposed by Ron Rivest as an improvement upon MD4. Collisions have been found for MD5, and the strength of MD5 is now less certain.

The Secure Hash Algorithm was developed by the National Security Agency (NSA) and given to the National Institute of Standards and Technology (NIST). The original version, often referred to as SHA or SHA-0, was published in 1993 as a Federal Information Processing Standard (FIPS 180). SHA contained a weakness that was later uncovered by the NSA, which led to a revised standards document (FIPS 180-1) that was released in 1995. This revised document describes the improved version, SHA-1, which for several years was the hash algorithm recommended by NIST. However, weaknesses started to appear and in 2017, a collision was found (see the discussion in [Section 11.1](#)). SHA-1 is now being replaced by a series of more secure versions called SHA-2. They still use the Merkle-Damgård construction. In the next section, we'll meet SHA-3, which uses a different construction.

The reader is warned that the discussion that follows is fairly technical and is provided in order to give the flavor of what happens inside a hash function.

The SHA-2 family consists of six algorithms: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. The last three digits indicate the number of bits in the output. We'll describe SHA-256. The other five are very similar.

SHA-256 produces a 256-bit hash and is built upon the same design principles as MD4, MD5, and SHA-1. These hash functions use an iterative procedure. Just as we did earlier, the original message M is broken into a set of fixed-size blocks, $M = M^{(1)} \parallel M^{(2)} \parallel \dots \parallel M^{(N)}$, where the last block is padded to fill out the block. The message blocks are then processed via a sequence of rounds that use a compression function h' that combines the current block and the result from the previous round. That is, we start with an initial value X_0 , and define $X_j = h'(X_{j-1}, M^{(j)})$. The final X_N is the message digest.

The trick behind building a hash function is to devise a good compression function. This compression function should be built in such a way as to make each input bit affect as many output bits as possible. One main difference between the SHA family and the MD family is that for SHA the input bits are used more often during the course of the hash function than they are for MD4 and MD5. This more conservative approach makes the design of SHA-1 and SHA-2 more secure than either MD4 or MD5, but also makes it a little slower.

In the description of the hash algorithm, we need the following operations on strings of 32 bits:

1. $X \wedge Y$ = bitwise “and”, which is bitwise multiplication mod 2, or bitwise minimum.

2. $X \vee Y$ = bitwise “or”, which is bitwise maximum.
3. $X \oplus Y$ = bitwise addition mod 2.
4. $\neg X$ changes 1s to 0s and 0s to 1s .
5. $X + Y$ = addition of X and Y mod 2^{32} , where X and Y are regarded as integers mod 2^{32} .
6. $R^n(X)$ = rotation of X to the right by n positions (the end wraps around to the beginning).
7. $S^n(X)$ = shift of X to the right by n positions, with the first n bits becoming 0s (so the bits at the end disappear and do not wrap around).

We also need the following functions that operate on 32-bit strings:

$$\begin{aligned}
 Ch(X, Y, Z) &= (X \wedge Y) \oplus (\neg X \wedge Z) \\
 Maj(X, Y, Z) &= (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) \\
 \Sigma_0(X) &= R^2(X) \oplus R^{13}(X) \oplus R^{22}(X) \\
 \Sigma_1(X) &= R^6(X) \oplus R^{11}(X) \oplus R^{25}(X) \\
 \sigma_0(X) &= R^7(X) \oplus R^{18}(X) \oplus S^3(X) \\
 \sigma_1(X) &= R^{17}(X) \oplus R^{19}(X) \oplus S^{10}(X).
 \end{aligned}$$

Define initial hash values $H_1^{(0)}, H_2^{(0)}, \dots, H_8^{(0)}$ as follows:

$$\begin{aligned}
 H_1^{(0)} &= 6A09E667 & H_2^{(0)} &= BB67AE85 & H_3^{(0)} &= 3C6EF372 & H_4^{(0)} &= A54FF53A \\
 H_5^{(0)} &= 510E527F & H_6^{(0)} &= 9B05688C & H_7^{(0)} &= 1F83D9AB & H_8^{(0)} &= 5BE0CD19
 \end{aligned}$$

The preceding are written in **hexadecimal notation**.

Each digit or letter represents a string of four bits:

$$0 = 0000, 1 = 0001, 2 = 0010, \dots, 9 = 1001,$$

$$A = 1010, B = 1011, \dots, F = 1111.$$

For example, BA1 equals

$$11 * 16^2 + 10 * 16^1 + 1 = 2977.$$

These initial hash values are obtained by using the first eight digits of the fractional parts of the square roots of the first eight primes, expressed as “decimals” in base 16.

See Exercise 7.

We also need sixty-four 32-bit words

$$K_0 = 428A2F98, \quad K_1 = 71374491, \quad \dots, \quad K_{63} = C67178f2.$$

They are the first eight hexadecimal digits of the fractional parts of the cube roots of the first 64 primes.

Padding and Preprocessing

SHA-256 begins by taking the original message and padding it with the bit 1 followed by a sequence of 0 bits. Enough 0 bits are appended to make the new message 64 bits short of the next highest multiple of 512 bits in length. Following the appending of 1 and 0s, we append the 64-bit representation of the length T of the message. (This restricts the messages to length less than $2^{64} \approx 10^{19}$ bits, which is not a problem in practice.)

For example, if the original message has 2800 bits, we add a 1 and 207 0s to obtain a new message of length $3008 = 6 \times 512 - 64$. Since $2800 = 101011110000_2$ in binary, we append fifty-two 0s followed by 101011110000 to obtain a message of length 3072. This is broken into six blocks of length 512.

Break the message with padding into N blocks of length 512:

$$M^{(1)} \parallel M^{(2)} \parallel \dots \parallel M^{(N)}.$$

The hash algorithm inputs these blocks one by one. In the algorithm, each 512-bit block $M^{(i)}$ is divided into sixteen 32-bit blocks:

$$M^{(i)} = M_0^{(i)} \parallel M_1^{(i)} \parallel \dots \parallel M_{15}^{(i)}.$$

The Algorithm

There are eight 32-bit registers, labeled a, b, c, d, e, f, g, h . These contain the intermediate hash values. The algorithm inputs a block of 512 bits from the message in Step 11, and in Steps 12 through 24, it stirs the bits of this block into a mix with the bits from the current intermediate hash values. After 64 iterations of this stirring, the algorithm produces an output that is added $(\text{mod } 2^{32})$ onto the previous intermediate hash values to yield the new hash values. After all of the blocks of the message have been processed, the final intermediate hash values give the hash of the message.

The basic building block of the algorithm is the set of operations that take place on the subregisters in Steps 15 through 24. They take the subregisters and operate on them using rotations, XORs, and other similar operations.

For more details on hash functions, and for some of the theory involved in their construction, see [Stinson], [Schneier], and [Menezes et al.].

Algorithm 3 The SHA-256 algorithm

- 1: **for** i from 1 to N **do**
 - ▷ This initializes the registers with the $(i - 1)$ st intermediate hash value
 - 2: $a \leftarrow H_1^{(i-1)}$
 - 3: $b \leftarrow H_2^{(i-1)}$
 - 4: $c \leftarrow H_3^{(i-1)}$
 - 5: $d \leftarrow H_4^{(i-1)}$
 - 6: $e \leftarrow H_5^{(i-1)}$
 - 7: $f \leftarrow H_6^{(i-1)}$

- 8: $g \leftarrow H_7^{(i-1)}$
- 9: $h \leftarrow H_8^{(i-1)}$
- 10: **for** k from 0 to 15 **do**
 - 11: $W_k \leftarrow M_k^{(i)}$ ▷ This is where the message blocks are entered.
 - 12: **for** j_1 from 16 to 63 **do**
 - 13: $W_{j_1} \leftarrow \sigma_1(W_{j_1-2}) + W_{j_1-7} + \sigma_0(W_{j_1-15}) + W_{j_1-16}$
 - 14: **for** j from 0 to 63 **do**
 - 15: $T_1 \leftarrow h + \Sigma_1(e) + Ch(e, f, g) + K_j + W_j$
 - 16: $T_2 \leftarrow \Sigma_0(a) + Maj(a, b, c)$
 - 17: $h \leftarrow g$
 - 18: $g \leftarrow f$
 - 19: $f \leftarrow e$
 - 20: $e \leftarrow d + T_1$
 - 21: $d \leftarrow c$
 - 22: $c \leftarrow b$
 - 23: $b \leftarrow a$
 - 24: $a \leftarrow T_1 + T_2$
 - 25: $H_1^{(i)} \leftarrow a + H_1^{(i-1)}$ ▷ These are the i th intermediate hash values
 - 26: $H_2^{(i)} \leftarrow b + H_2^{(i-1)}$
 - 27: $H_3^{(i)} \leftarrow c + H_3^{(i-1)}$
 - 28: $H_4^{(i)} \leftarrow d + H_4^{(i-1)}$
 - 29: $H_5^{(i)} \leftarrow e + H_5^{(i-1)}$
 - 30: $H_6^{(i)} \leftarrow f + H_6^{(i-1)}$
 - 31: $H_7^{(i)} \leftarrow g + H_7^{(i-1)}$
 - 32: $H_8^{(i)} \leftarrow h + H_8^{(i-1)}$
 - 33:

$$H(m) = H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)} \parallel H_8^{(N)}$$

- 34: **return** $H(m)$

11.5 SHA-3/Keccak

In 2006, NIST announced a competition to produce a new hash function to serve alongside SHA-2. The new function was required to be at least as secure as SHA-2 and to have the same four output possibilities. Fifty-one entries were submitted, and in 2012, **Keccak** was announced as the winner. It was certified as a standard by NIST in 2012 in FIPS-202. (The name is pronounced “ketchak”. It has been suggested that the name is related to “Kecak,” a type of Balinese dance. Perhaps the movement of the dancers is analogous to the movement of the bits during the algorithm.) This algorithm became the hash function SHA-3.

The SHA-3 algorithm was developed by Guido Bertoni, Joan Daemen, and Gilles Van Assche from STMicroelectronics and Michaël Peeters from NXP Semiconductors. It differs from the Merkle-Damgård construction and is based on the theory of **Sponge**

Functions. The idea is that the first part of the algorithm absorbs the message, and then the hash value is squeezed out. Here is how it works. The **state** of the machine is a string of b bits, which is fed to a function f that takes an input of b bits and outputs a string of b bits, thus producing a new state of the machine. In contrast to the compression functions in the Merkle-Damgård construction, the function f is a one-to-one function. Such a function could not be used in the Merkle-Damgård situation since the number of input bits (from M_i and the previous step) is greater than the number of output bits. But the different construction in the present case allows it.

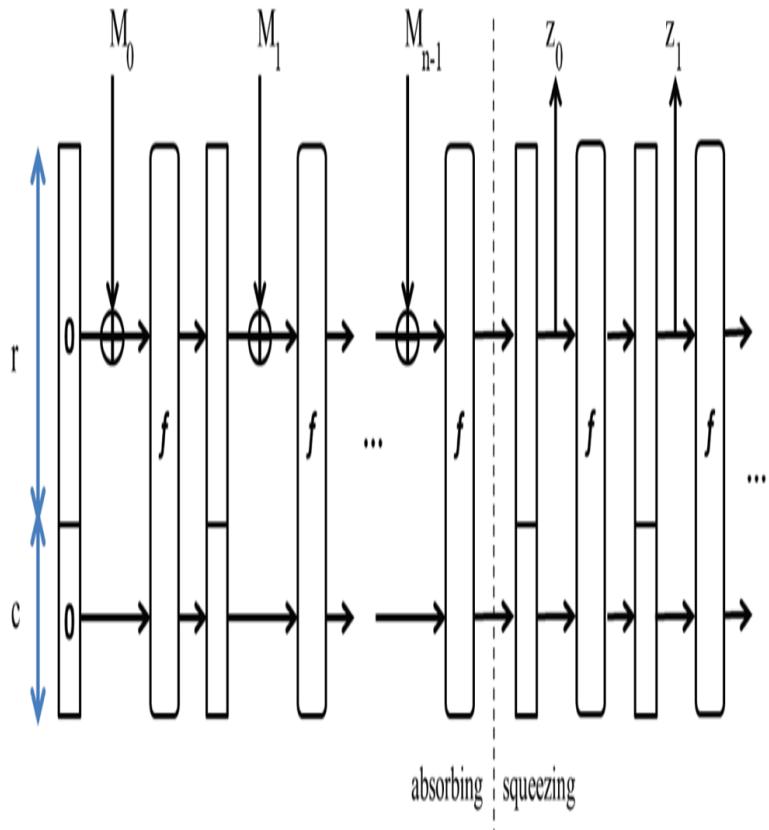
Parameters r (“the rate”) and c (“the capacity”) are chosen so that $r + c = b$. The message (written in

binary) is padded so that its length is a multiple of r , then is broken into n blocks of length r :

$$M = M_0 || M_1 || M_2 || \cdots || M_{n-1}.$$

To start, the state is initialized to all os. The absorption stage is the first part of [Figure 11.2](#).

Figure 11.2 A Sponge Function



[Figure 11.2 Full Alternative Text](#)

After the absorption is finished, the hash value is squeezed out: r bits are output and truncated to the d bits that are used as the hash value. This is the second part of [Figure 11.2](#).

Producing a SHA-3 hash value requires only one squeeze. However, the algorithm can also be used with multiple squeezes to produce arbitrarily long

pseudorandom bitstrings. When it is used this way, it is often called SHAKE (= Secure Hash Algorithm with Keccak).

The “length extension” and collision-based attacks of [Section 11.3](#) are less likely to succeed. Suppose two messages yield the same hash value. This means that when the absorption has finished and the squeezing stage is starting, there are d bits of the state that agree with these bits for the other message. But there are at least c bits that are not output, and there is no reason that these c bits match. If, instead of starting the squeezing, you do another round of absorption, the differing c bits will cause the subsequent states and the outputted hash values to differ. In other words, there are at least 2^c possible internal states for any given d -bit output H .

SHA-3 has four different versions, named SHA3-224, SHA3-256, SHA3-384, and SHA3-512. For SHA3- m , the m denotes the security level, measured in bits. For example, SHA3-256 is expected to require around 2^{256} operations to find a collision or a preimage. Since $2^{256} \approx 10^{77}$, this should be impossible well into the future. The parameters are taken to be

$$b = 1600, \quad d = m, \quad c = 2m, \quad r = 1600 - c.$$

For SHA3-256, these are

$$b = 1600, \quad d = 256, \quad c = 512, \quad r = 1088.$$

The same function f is used in all versions, which means that it is easy to change from one security level to another. Note that there is a trade-off between speed and security. If the security parameter m is increased, then r decreases, so the message is read slower, since it is read in blocks of r bits.

In the following, we concentrate on SHA3-256. The other versions are obtained by suitably varying the parameters.

For more details, see [FIPS 202].

The Padding. We start with a message M . The message is read in blocks of $r = 1088$ bits, so we want the message to have length that is a multiple of 1088. But first the message is padded to $M||01$. This is for “domain separation.” There are other uses of the Keccak algorithm such as SHAKE (mentioned above), and for these other purposes, M is padded differently, for example with 1111. This initial padding makes it very likely that using M in different situations yields different outputs. Next, “10*1 padding” is used. This means that first a 1 is appended to M_011 to yield $M||011$. Then sufficiently many 01 s are appended to make the total length one less than a multiple of 1088. Finally, a 1 is appended. We can now divide the result into blocks of length 1088.

Why are these choices made for the padding? Why not simply append enough 0s to get the desired length?

Suppose that $M_1 = 1010111$. Then

$M_1||10 = 101011110$. Now append 1079 zeros to get the block to be hashed. If $M_2 = 1010$ is being used in SHAKE, then $M_2||1111$ is padded with 1080 zeros to yield the same block. This means that the outputs for M_1 and M_2 are equal. The padding is designed to avoid all such situations.

Absorption and Squeezing. From now on, we assume that the padding has been done and we have N blocks of length 1088:

$$M_0||M_1||\dots||M_{N-1}.$$

The absorption now proceeds as in [Figure 11.2](#) (we describe the function f later).

1. The initial state S is a string of os s of length 1600.
2. For $j = 0$ to $N - 1$, let $S = f(S \oplus (M_j||0^c))$, where 0^c denotes a string of os of length $c = 512$. What this does is XOR the message block M_j with the first $r = 1088$ bits of S , and then

apply the function f . This yields an updated state S , which is modified during each iteration of the index j .

3. Return S .

The squeezing now proceeds as in [Figure 11.2](#):

1. Input S and let Z be the empty string.
2. While $\text{Length}(Z) < \text{td}$ (where $d = 256$ is the output size)
 1. Let $Z = Z || \text{Trunc}_r(S)$, where $\text{Trunc}_r(S)$ denotes the first $r = 1088$ bits of S .
 2. $S = f(S)$.
3. Return $\text{Trunc}_d(Z)$.

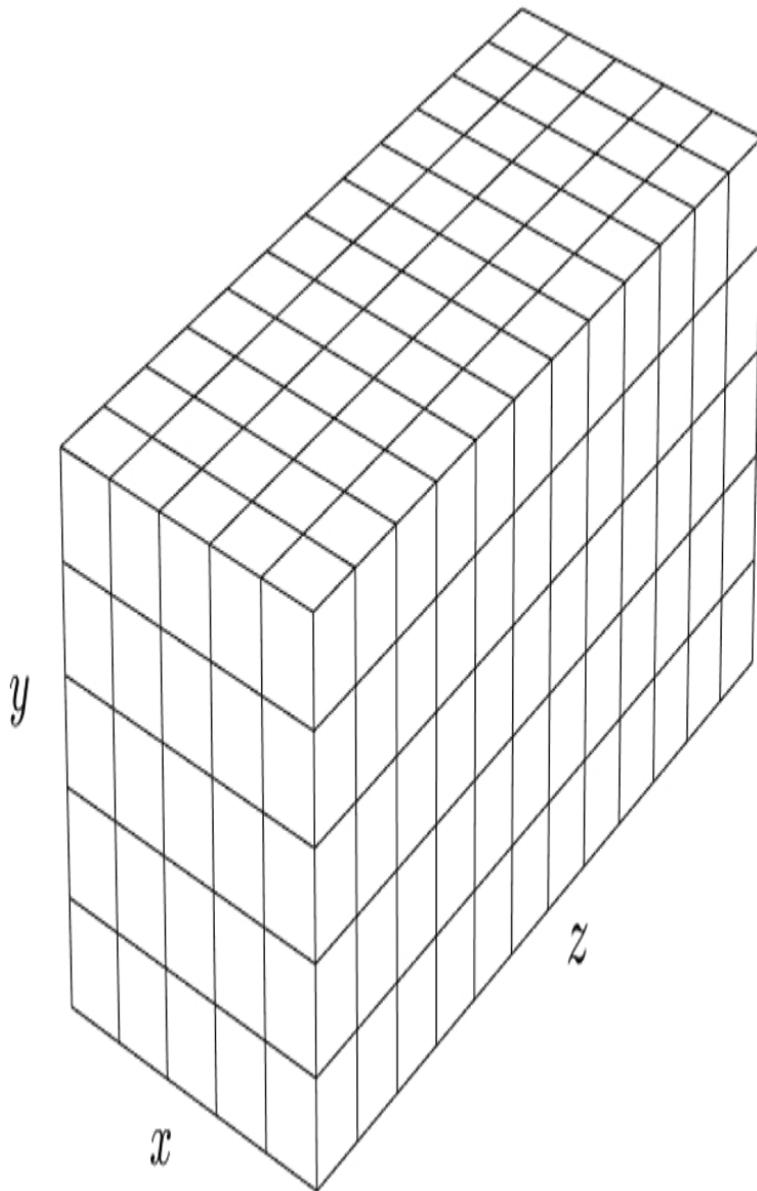
The bitstring $\text{Trunc}_{256}(Z)$ is the 256-bit hash value $\text{SHA-3}(M)$. For the hash value, we need only one squeeze to obtain Z . But the algorithm could also be used to produce a much longer pseudorandom bitstring, in which case several squeezes might be needed.

The function f . The main component of the algorithm is the function f , which we now describe. The input to f is the 1600-bit state of the machine

$$S = S[0] || S[1] || S[2] || \dots || S[1599],$$

where each $S[j]$ is a bit. It's easiest to think of these bits as forming a three-dimensional $5 \times 5 \times 64$ array $A[x, y, z]$ with coordinates (x, y, z) satisfying

$$0 \leq x \leq 4, \quad 0 \leq y \leq 4, \quad 0 \leq z \leq 63.$$



A “column” consists of the five bits with fixed x, z . A “row” consists of the five bits with fixed y, z . A “lane” consists of the 64 bits with fixed x, y .

When we write “for all x, z ” we mean for $0 \leq x \leq 4$ and $0 \leq z \leq 63$, and similarly for other combinations of x, y, z .

The correspondence between S and A is given by

$$A[x, y, z] = S[64(5y + x) + z]$$

for all x, y, z . For example, $A[1, 2, 3] = S[707]$. The ordering of the indices could be described as

“lexicographic” order using y, x, z (not x, y, z), since the index of S corresponding to x_1, y_1, z_1 is smaller than the index for x_2, y_2, z_2 if $y_1x_1z_1$ precedes $y_2x_2z_2$ in “alphabetic order.”

The coordinates x, y are taken to be numbers mod 5, and z is mod 64. For example $A[7, -1, 86]$ is taken to be $A[2, 4, 22]$, since $7 \equiv 2 \pmod{5}$, $-1 \equiv 4 \pmod{5}$, and $86 \equiv 22 \pmod{64}$.

The computation of f proceeds in several steps. The steps receive the array A as input and they output a modified array A' to replace A .

The following steps I through V are repeated for $i = 0$ to $i = 23$:

1. The first step XORs the bits in a column with the parities of two nearby columns.

1. For all x, z , let

$$C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z].$$

This gives the “parity” of the bitstring formed by the five bits in the x, z column.

2. For all x, z , let

$$D[x, z] = C[x - 1, z] \oplus C[x + 1, z - 1].$$

3. For all x, y, z , let $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$.

2. The second step rotates the 64 bits in each lane by an amount depending on the lane:

1. For all z , let $A'[0, 0, z] = A[0, 0, z]$.

2. Let $(x, y) = (1, 0)$.

3. For $t = 0$ to 23

1. For all z , let

$$A'[x, y, z] = A[x, y, z - (t + 1)(t + 2)/2]$$

.

2. Let $(x, y) = (y, 2x + 3y)$
 4. Return A' .
- For example, consider the bits with coordinates of the form $(1, 0, z)$. They are handled by the case $t = 0$ and we have $A'[1, 0, z] = A[1, 0, z - 1]$, so the bits in this lane are rotated by one position. Then, in Step 3(b), $(x, y) = (1, 0)$ is changed to $(0, 2)$ for the iteration with $t = 1$. We have $A'[0, 2, z] = A[0, 2, z - 3]$, so this lane is rotated by three positions. Then (x, y) is changed to $(2, 6)$, which is reduced mod 5 to $(2, 1)$, and we pass to $t = 2$, which gives a rotation by six (the rotations are by what are known as “triangular numbers”). After $t = 23$, all of the lanes have been rotated.
3. The third step rearranges the positions of the lanes:
 1. For all x, y, z , let $A'[x, y, z] = A[x + 3y, x, z]$.
- Again, the coordinate $x + 3y$ should be reduced mod 5.
4. The next step is the only nonlinear step in the algorithm. It XORs each bit with an expression formed from two other bits in its row.
 1. For all x, y, z , let
 2. $A'[x, y, z] = A[x, y, z] \oplus (A[x + 1, y, z] \oplus 1)(A[x + 2, y, z])$.
 3. The multiplication is multiplying two binary bits, hence is the same as the *AND* operator.
5. Finally, some bits in the $(0, 0)$ lane are modified.
 1. For all x, y, z , let $A'[x, y, z] = A[x, y, z]$.
 2. Set $RC = 0^{24}$.
 3. For $j = 0$ to 6, let $RC[2^j - 1] = rc(j + 7i)$, where rc is an auxiliary function defined below.
 4. For all z , let $A'[0, 0, z] = A'[0, 0, z] \oplus RC[z]$.
 5. Return A' .

After I through V are completed for one value of i , the next value of i is used and I through V are repeated for the new i , through $i = 23$. The final output is the new array A , which yields a new bitstring S of length 1600.

This completes the description of the function f , except that we still need to describe the auxiliary function rc .

The function rc takes an integer $t \bmod 255$ as input and outputs a bit according to the following algorithm:

1. If $t \equiv 0 \pmod{255}$, return 1. Else
2. $R = 10000000$
3. For $k = 1$ to $t \bmod 255$
 1. Let $R = 0||R$
 2. Let $R[0] = R[0] \oplus R[8]$
 3. Let $R[4] = R[4] \oplus R[8]$
 4. Let $R[5] = R[5] \oplus R[8]$
 5. Let $R[6] = R[6] \oplus R[8]$
 6. Let $R = \text{Trunc}_8(R)$.
4. Return $R[0]$.

The bit that is outputted is $rc(t)$.

11.6 Exercises

1. Let p be a prime and let α be an integer with $p \nmid \alpha$. Let $h(x) \equiv \alpha^x \pmod{p}$. Explain why $h(x)$ is not a good cryptographic hash function.
2.
 1. Alice claims that she knows who will win the next World Cup. She takes the name of the team, T , and encrypts it with a one-time pad K , and sends $C = T \oplus K$ to Bob. After the World Cup is finished, Alice reveals K , and Bob computes $T = C \oplus K$ to determine Alice's guess. Why should Bob not believe that Alice actually guessed the correct team, even if $T = C \oplus K$ is correct?
 2. To keep Alice from changing K , Bob requires Alice to send not only $C = T \oplus K$ but also $H(K)$, where H is a good cryptographic hash function. How does the use of the hash function convince Bob that Alice is not changing K ?
 3. In the procedure in (b), Bob receives C and $H(K)$. Show how he can determine Alice's prediction, without needing Alice to send K ? (*Hint:* There are fewer than 100 teams T that could win the World Cup.)
3. Let $n = pq$ be the product of two distinct large primes and let $h(x) = x^2 \pmod{n}$.
 1. Why is h preimage resistant? (Of course, there are some values, such as 1, 4, 9, 16, \dots for which it is easy to find a preimage. But usually it is difficult.)
 2. Why is h not strongly collision resistant?
4. Let $H(x)$ be a cryptographic hash function. Nelson tries to make new hash functions.
 1. He takes a large prime p and a primitive root α for p . For an input x , he computes $\beta \equiv \alpha^x \pmod{p}$, then sets $H_2(x) = H(\beta)$. The function H_2 is not fast enough to be a hash function. Find one other property of hash functions that fails for H_2 and one that holds for H_2 , and justify your answers.
 2. Since his function in part (a) is not fast enough, Nelson tries using $H_3 = H(x \bmod p)$. This is very fast. Find

one other property of hash functions that holds for H_3 and one that fails for H_3 , and justify your answers.

5. Suppose a message m is divided into blocks of length 160 bits: $m = M_1 || M_2 || \dots || M_\ell$. Let $h(x) = M_1 \oplus M_2 \oplus \dots \oplus M_\ell$. Which of the properties (1), (2), (3) for a hash function does h satisfy and which does it not satisfy? Justify your answers.

6. One way of storing and verifying passwords is the following. A file contains a user's login id plus the hash of the user's password. When the user logs in, the computer checks to see if the hash of the password is the same as the hash stored in the file. The password is not stored in the file. Assume that the hash function is a good cryptographic hash function.

1. Suppose Eve is able to look at the file. What property of the hash function prevents Eve from finding a password that will be accepted as valid?
2. When the user logs in, and the hash of the user's password matches the hash stored in the file, what property of the hash function says that the user probably entered the correct password? (*Hint:* Your answers to (a) and (b) should not be the same.)

7. The initial values $H_k^{(0)}$ in SHA-256 are extracted from the hexadecimal expansions of the fractional parts of the square roots of the first eight primes. Here is what that means.

1. Compute $\lfloor 2^{32}(\sqrt{2} - 1) \rfloor$ and write the answer in hexadecimal. The answer should be $H_1^{(0)}$.
2. Do a similar computation with $\sqrt{2}$ replaced by $\sqrt{3}$, $\sqrt{5}$, and $\sqrt{19}$ and compare with the appropriate values of $H_k^{(0)}$.

8. Alice and Bob (and no one else) share a key K . Each time that Alice wants to make sure that she is communicating with Bob, she sends him a random string S of 100 bits. Bob computes $B = H(S||K)$, where H is a good cryptographic hash function, and sends B to Alice. Alice computes $H(S||K)$. If this matches what Bob sent her, she is convinced that she is communicating with Bob.

1. What property of H convinces Alice that she is communicating with Bob?
2. Suppose Alice's random number generator is broken and she sends the same S each time she communicates with anyone. How can Eve (who doesn't know K , but who

intercepts all communications between Alice and Bob) convince Alice that she is Bob?

9.
 1. Show that neither of the two hash functions of [Section 11.2](#) is preimage resistant. That is, given an arbitrary y (of the appropriate length), show how to find an input x whose hash is y .
 2. Find a collision for each of the two hash functions of [Section 11.2](#).
10. An unenlightened professor asks his students to memorize the first 1000 digits of π for the exam. To grade the exam, he uses a 100-digit cryptographic hash function H . Instead of carefully reading the students' answers, he hashes each of them individually to obtain binary strings of length 100. Your score on the exam is the number of bits of the hash of your answer that agree with the corresponding bits of the hash of the correct answer.
 1. If someone gets 100% on the exam, why is the professor confident that the student's answer is correct?
 2. Suppose each student gets every digit of π wrong (a very unlikely occurrence!), and they all have different answers. Approximately what should the average on the exam be?
11. A bank in Tokyo is sending a terabyte of data to a bank in New York. There could be transmission errors. Therefore, the bank in Tokyo uses a cryptographic hash function H and computes the hash of the data. This hash value is sent to the bank in New York. The bank in New York computes the hash of the data received. If this matches the hash value sent from Tokyo, the New York bank decides that there was no transmission error. What property of cryptographic hash functions allows the bank to decide this?
12. (Thanks to Danna Doratotaj for suggesting this problem.)
 1. Let H_1 map 256-bit strings to 256-bit strings by $H_1(x) = x$. Show that H_1 is not preimage resistant but it is collision resistant.
 2. Let H be a good cryptographic hash function with a 256-bit output. Define a map H_2 from binary strings of arbitrary length to binary strings of length 257, as follows. If $0 \leq x < 2^{256}$, let $H_2(x) = x||$ (that is, x with appended). If $x \geq 2^{256}$, let $H_2(x) = H(x)||1$. Show that H_2 is collision resistant, and show that if y is a randomly chosen binary string of length 257, then the probability is at least 50% that you can easily find x with $H(x) = y$.

The functions H_1 and H_2 show that collision resistance does not imply preimage resistance quite as obviously as one might suspect.

13. Show that the computation of rc in Keccak (see the end of [Section 11.5](#)) can be given by an LFSR.

14. Let $Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$ be the function on 32-bit strings in the description of SHA-2.

1. Suppose that the first bit of X and the first bit of Y are 1 and the first bit of Z is arbitrary. Show that the first bit of $Maj(X, Y, Z)$ is 1.

2. Suppose that the first bit of X and the first bit of Y are 0 and the first bit of Z is arbitrary. Show that the first bit of $Maj(X, Y, Z)$ is 0.

This shows that Maj gives the bitwise majority (for 0 vs. 1) for the strings X, Y, Z .

15. Let H be an iterative hash function that operates successively on input blocks of 512 bits. In particular, there is a compression function h and an initial value IV . The hash of a message $M_1 \parallel M_2$ of 1024 bits is computed by $X_1 = h(IV, M_1)$, and $H(M_1 \parallel M_2) = h(X_1, M_2)$. Suppose we have found a collision $h(IV, x_1) = h(IV, x_2)$ for some 512-bit blocks x_1 and x_2 . Choose distinct primes p_1 and p_2 , each of approximately 240 bits. Regard x_1 and x_2 as numbers between 0 and 2^{512} .

1. Show that there exists an x_0 with $0 \leq x_0 < p_1 p_2$ such that

$$x_0 + 2^{512}x_1 \equiv 0 \pmod{p_1} \text{ and } x_0 + 2^{512}x_2 \equiv 0 \pmod{p_2}.$$

2. Show that if $0 \leq k < 2^{30}$, then

$q_1 = (x_0 + 2^{512}x_1 + kp_1 p_2)/p_1$ is approximately 2^{784} , and similarly for

$q_2 = (x_0 + 2^{512}x_2 + kp_1 p_2)/p_2$. (Assume that x_1 and x_2 are approximately 2^{512} .)

3. Use the Prime Number Theorem (see [Section 3.1](#)) to show that the probability that q_1 is prime is approximately $1/543$ and the probability that both q_1 and q_2 are prime is about $1/300000$.

4. Show that it is likely that there is some k with $0 \leq k < 2^{30}$ such that both q_1 and q_2 are primes.

5. Show that $n_1 = p_1 q_1$ and $n_2 = p_2 q_2$ satisfy $H(n_1) = H(n_2)$.

This method of producing two RSA moduli with the same hash values is based on the method of [Lenstra et al.] for using a

collision to produce two X.509 certificates with the same hashes. The method presented here produces moduli $n = pq$ with p and q of significantly different sizes (240 bits and 784 bits), but an adversary does not know this without factoring n .

Chapter 12 Hash Functions: Attacks and Applications

12.1 Birthday Attacks

If there are 23 people in a room, the probability is slightly more than 50% that two of them have the same birthday. If there are 30, the probability is around 70%. This might seem surprising; it is called the **birthday paradox**. Let's see why it's true. We'll ignore leap years (which would slightly lower the probability of a match) and we assume that all birthdays are equally likely (if not, the probability of a match would be slightly higher).

Consider the case of 23 people. We'll compute the probability that they all have different birthdays. Line them up in a row. The first person uses up one day, so the second person has probability $(1 - 1/365)$ of having a different birthday. There are two days removed for the third person, so the probability is $(1 - 2/365)$ that the third birthday differs from the first two. Therefore, the probability of all three people having different birthdays is $(1 - 1/365)(1 - 2/365)$. Continuing in this way, we see that the probability that all 23 people have different birthdays is

$$(1 - \frac{1}{365})(1 - \frac{2}{365}) \cdots (1 - \frac{22}{365}) = .493.$$

Therefore, the probability of at least two having the same birthday is

$$1 - .493 = .507.$$

One way to understand the preceding calculation intuitively is to consider the case of 40 people. If the first

30 have a match, we're done, so suppose the first 30 have different birthdays. Now we have to choose the last 10 birthdays. Since 30 birthdays are already chosen, we have approximately a 10% chance that a randomly chosen birthday will match one of the first 30. And we are choosing 10 birthdays. Therefore, it shouldn't be too surprising that we get a match. In fact, the probability is 89% that there is a match among 40 people.

More generally, suppose we have N objects, where N is large. There are r people, and each chooses an object (with replacement, so several people could choose the same one). Then

$$\text{Prob}(\text{there is a match}) \approx 1 - e^{-r^2/2N}.$$

(12.1)

Note that this is only an approximation that holds for large N ; for small n it is better to use the above product and obtain an exact answer. In [Exercise 12](#), we derive this approximation. Choosing $r^2/2N = \ln 2$, we find that if $r \approx 1.177\sqrt{N}$, then the probability is 50% that at least two people choose the same object.

To summarize, if there are N possibilities and we have a list of length \sqrt{N} , then there is a good chance of a match. If we want to increase the chance of a match, we can make the list have length $2\sqrt{N}$ or $5\sqrt{N}$. The main point is that a length of a constant times \sqrt{N} (instead of something like N) suffices.

For example, suppose we have 40 license plates, each ending in a three-digit number. What is the probability that two of the license plates end in the same three digits? We have $N = 1000$, the number of possible three-digit numbers, and $r = 40$, the number of license plates under consideration. Since

$$\frac{r^2}{2N} = .8,$$

the approximate probability of a match is

$$1 - e^{-8} = .551,$$

which is more than 50%. We stress that this is only an approximation. The correct answer is obtained by calculating

$$1 - \left(1 - \frac{1}{1000}\right) \left(1 - \frac{2}{1000}\right) \cdots \left(1 - \frac{39}{1000}\right) = .546.$$

The next time you are stuck in traffic (and have a passenger to record numbers), check out this prediction.

But what is the probability that one of these 40 license plates has the same last three digits as yours (assuming that yours ends in three digits)? Each plate has probability $1 - 1/1000$ of not matching yours, so the probability is $(1 - 1/1000)^{40} = .961$ that none of the 40 plates matches your plate. The reason the birthday paradox works is that we are not just looking for matches between one fixed plate, such as yours, and the other plates. We are looking for matches between any two plates in the set, so there are many more opportunities for matches.

For more examples, see Examples 36 and 37 in the Computer Appendices.

The applications of these ideas to cryptology require a slightly different setup. Suppose there are two rooms, each with 30 people. What is the probability that someone in the first room has the same birthday as someone in the second room? More generally, suppose there are N objects and there are two groups of r people. Each person from each group selects an object (with replacement). What is the probability that someone from the first group chooses the same object as someone from the second group? In this case,

$$\text{Prob}(\text{there is a match}) \approx 1 - e^{-r^2/N}.$$

(12.2)

If $\lambda = r^2/N$, then the probability of exactly i matches is approximately $\lambda^i e^{-\lambda} / i!$. An analysis of this problem, with generalizations, is given in [Girault et al.]. Note that the present situation differs from the earlier problem of finding a match in one set of r people. Here, we have two sets of r people, so a total of $2r$ people. Therefore, the probability of a match in this set is approximately $1 - e^{-2r^2/N}$. But around half of the time, these matches are between members of the same group, and half the time the matches are the desired ones, namely, between the two groups. The precise effect is to cut the probability down to $1 - e^{-r^2/N}$.

Again, if there are N possibilities and we have two lists of length \sqrt{N} , then there is a good chance of a match. Also, if we want to increase the chance of a match, we can make the lists have length $2\sqrt{N}$ or $5\sqrt{N}$. The main point is that a length of a constant times \sqrt{N} (instead of something like N) suffices.

For example, if we take $N = 365$ and $r = 30$, then

$$\lambda = 30^2/365 = 2.466.$$

Since $1 - e^{-\lambda} = .915$, there is approximately a 91.5% probability that someone in one group of 30 people has the same birthday as someone in a second group of 30 people.

The birthday attack can be used to find collisions for hash functions if the output of the hash function is not sufficiently large. Suppose that h is an n -bit hash function. Then there are $N = 2^n$ possible outputs. Make a list $h(x)$ for approximately $r = \sqrt{N} = 2^{n/2}$ random choices of x . Then we have the situation of $r \approx \sqrt{N}$ “people” with N possible “birthdays,” so there is a good chance of having two values x_1 and x_2 with the same hash value. If we make the list longer, for example

$r = 10 \cdot 2^{n/2}$ values of x , the probability becomes very high that there is a match.

Similarly, suppose we have two sets of inputs, S and T . If we compute $h(s)$ for approximately \sqrt{N} randomly chosen $s \in S$ and compute $h(t)$ for approximately \sqrt{N} randomly chosen $t \in T$, then we expect some value $h(s)$ to be equal to some value $h(t)$. This situation will arise in an attack on signature schemes in Chapter 13, where S will be a set of good documents and T will be a set of fraudulent documents.

If the output of the hash function is around $n = 60$ bits, the above attacks have a high chance of success. It is necessary to make lists of length approximately $2^{n/2} = 2^{30} \approx 10^9$ and to store them. This is possible on most computers. However, if the hash function outputs 256-bit values, then the lists have length around $2^{128} \approx 10^{38}$, which is too large, both in time and in memory.

12.1.1 A Birthday Attack on Discrete Logarithms

Suppose we are working with a large prime p and want to evaluate $L_\alpha(h)$. In other words, we want to solve $\alpha^x \equiv h \pmod{p}$. We can do this with high probability by a birthday attack.

Make two lists, both of length around \sqrt{p} :

1. The first list contains numbers $\alpha^k \pmod{p}$ for approximately \sqrt{p} randomly chosen values of k .
2. The second list contains numbers $h\alpha^{-\ell} \pmod{p}$ for approximately \sqrt{p} randomly chosen values of ℓ .

There is a good chance that there is a match between some element on the first list and some element on the second list. If so, we have

$$\alpha^k \equiv h\alpha^{-\ell}, \text{ hence } \alpha^{k+\ell} \equiv h \pmod{p}.$$

Therefore, $x \equiv k + \ell \pmod{p-1}$ is the desired discrete logarithm.

Let's compare this method with the Baby Step, Giant Step (BSGS) method described in [Section 10.2](#). Both methods have running time and storage space proportional to \sqrt{p} . However, the BSGS algorithm is **deterministic**, which means that it is guaranteed to produce an answer. The birthday algorithm is **probabilistic**, which means that it probably produces an answer, but this is not guaranteed. Moreover, there is a computational advantage to the BSGS algorithm. Computing one member of a list from a previous one requires one multiplication (by α or by α^{-N}). In the birthday algorithm, the exponent k is chosen randomly, so α^k must be computed each time. This makes the algorithm slower. Therefore, the BSGS algorithm is somewhat superior to the birthday method.

12.2 Multicollisions

In this section, we show that the iterative nature of hash algorithms based on the Merkle-Damgård construction makes them less resistant than expected to finding multicollisions, namely inputs x_1, \dots, x_n all with the same hash value. This was pointed out by Joux [Joux], who also gave implications for properties of concatenated hash functions, which we discuss below.

Suppose there are r people and there are N possible birthdays. It can be shown that if $r \approx N^{(k-1)/k}$, then there is a good chance of at least k people having the same birthday. In other words, we expect a k -collision. If the output of a hash function is random, then we expect that this estimate holds for k -collisions of hash function values. Namely, if a hash function has n -bit outputs, hence $N = 2^n$ possible values, and if we calculate $r = 2^{n(k-1)/k}$ values of the hash function, we expect a k -collision. However, in the following, we'll show that often we can obtain collisions much more easily.

In many hash functions, for example, SHA-256, there is a compression function f that operates on inputs of a fixed length. Also, there is a fixed initial value IV . The message is padded to obtain the desired format, then the following steps are performed:

1. Split the message M into blocks M_1, M_2, \dots, M_ℓ .
2. Let H_0 be the initial value IV .
3. For $i = 1, 2, \dots, \ell$, let $H_i = f(H_{i-1}, M_i)$.
4. Let $H(M) = H_\ell$.

In SHA-256, the compression function is described in Section 11.4. For each iteration, it takes a 256-bit input

from the preceding iteration along with a message block M_i of length 512 and outputs a new string of length 256.

Suppose the output of the function f , and therefore also of the hash function H , has n bits. A birthday attack can find, in approximately $2^{n/2}$ steps, two blocks m_0 and m'_0 such that $f(H_0, m_0) = f(H_0, m'_0)$. Let $h_1 = f(H_0, m_0)$. A second birthday attack finds blocks m_1 and m'_1 with $f(h_1, m_1) = f(h_1, m'_1)$. Continuing in this manner, we let

$$h_i = f(h_{i-1}, m_{i-1})$$

and use a birthday attack to find m_i and m'_i with

$$f(h_i, m_i) = f(h_i, m'_i).$$

This process is continued until we have t pairs of blocks $m_0, m'_0, m_1, m'_1, \dots, m_{t-1}, m'_{t-1}$, where t is some integer to be determined later.

We claim that each of the 2^t messages

$$\begin{aligned} & m_0 \parallel m_1 \parallel \cdots \parallel m_{t-1} \\ & m'_0 \parallel m_1 \parallel \cdots \parallel m_{t-1} \\ & m_0 \parallel m'_1 \parallel \cdots \parallel m_{t-1} \\ & m'_0 \parallel m'_1 \parallel \cdots \parallel m_{t-1} \\ & \quad \dots \dots \dots \\ & m'_0 \parallel m_1 \parallel \cdots \parallel m'_{t-1} \\ & m_0 \parallel m'_1 \parallel \cdots \parallel m'_{t-1} \\ & m'_0 \parallel m'_1 \parallel \cdots \parallel m'_{t-1} \end{aligned}$$

(all possible combinations with m_i and m'_i) has the same hash value. This is because of the iterative nature of the hash algorithm. At each calculation

$h_i = f(m, h_{i-1})$, the same value h_i is obtained whether $m = m_{i-1}$ or $m = m'_{i-1}$. Therefore, the output of the function f during each step of the hash algorithm is independent of whether an m_{i-1} or an m'_{i-1} is used. Therefore, the final output of the hash algorithm is the same for all messages. We thus have a 2^t -collision.

This procedure takes approximately $t 2^{n/2}$ steps and has an expected running time of approximately a constant times $tn 2^{n/2}$ (see [Exercise 13](#)). Let $t = 2$, for example. Then it takes only around twice as long to find four messages with same hash value as it took to find two messages with the same hash. If the output of the hash function were truly random, rather than produced, for example, by an iterative algorithm, then the above procedure would not work. The expected time to find four messages with the same hash would then be approximately $2^{3n/4}$, which is much longer than the time it takes to find two colliding messages. Therefore, it is easier to find multicollisions with an iterative hash algorithm.

An interesting consequence of the preceding discussion relates to attempts to improve hash functions by concatenating their outputs. Suppose we have two hash functions H_1 and H_2 . Before [Joux] appeared, the general wisdom was that the concatenation

$$H(M) = H_1(M) \parallel H_2(M)$$

should be a significantly stronger hash function than either H_1 or H_2 individually. This would allow people to use somewhat weak hash functions to build much stronger ones. However, it now seems that this is not the case. Suppose the output of H_i has n_i bits. Also, assume that H_1 is calculated by an iterative algorithm, as in the preceding discussion. No assumptions are needed for H_2 . We may even assume that it is a random oracle, in the sense of [Section 12.3](#). In time approximately

$\frac{1}{2} n_2 n_1 2^{n_1/2}$, we can find $2^{n_2/2}$ messages that all have the same hash value for H_1 . We then compute the value of H_2 for each of these $2^{n_2/2}$ messages. By the birthday paradox, we expect to find a match among these values of H_2 . Since these messages all have the same H_1 value, we have a collision for $H_1 \parallel H_2$. Therefore, in time

proportional to $\frac{1}{2}n_2n_12^{n_1/2} + n_22^{n_2/2}$ (we'll explain this estimate shortly), we expect to be able to find a collision for $H_1 \parallel H_2$. This is not much longer than the time a birthday attack takes to find a collision for the longer of H_1 and H_2 , and is much faster than the time $2^{(n_1+n_2)/2}$ that a standard birthday attack would take on this concatenated hash function.

How did we get the estimate $\frac{1}{2}n_2n_12^{n_1/2} + n_22^{n_2/2}$ for the running time? We used $\frac{1}{2}n_2n_12^{n_1/2}$ steps to get the $2^{n_2/2}$ messages with the same H_1 value. Each of these messages consisted of n_2 blocks of a fixed length. We then evaluated H_2 for each of these messages. For almost every hash function, the evaluation time is proportional to the length of the input. Therefore, the evaluation time is proportional to n_2 for each of the $2^{n_2/2}$ messages that are given to H_2 . This gives the term $n_22^{n_2/2}$ in the estimated running time.

12.3 The Random Oracle Model

Ideally, a hash function is indistinguishable from a random function. The random oracle model, introduced in 1993 by Bellare and Rogaway [Bellare-Rogaway], gives a convenient method for analyzing the security of cryptographic algorithms that use hash functions by treating hash functions as random oracles.

A random oracle acts as follows. Anyone can give it an input, and it will produce a fixed length output. If the input has already been asked previously by someone, then the oracle outputs the same value as it did before. If the input is not one that has previously been given to the oracle, then the oracle gives a randomly chosen output. For example, it could flip n fair coins and use the result to produce an n -bit output.

For practical reasons, a random oracle cannot be used in most cryptographic algorithms; however, assuming that a hash function behaves like a random oracle allows us to analyze the security of many cryptosystems that use hash functions.

We already made such an assumption in [Section 12.1](#). When calculating the probability that a birthday attack finds collisions for a hash function, we assumed that the output of the hash function is randomly and uniformly distributed among all possible outcomes. If this is not the case, so the hash function has some values that tend to occur more frequently than others, then the probability of finding collisions is somewhat higher (for example, consider the extreme case of a really bad hash function that, with high probability, outputs only one value). Therefore, our estimate for the probability of collisions

really only applies to an idealized setting. In practice, the use of actual hash functions probably produces very slightly more collisions.

In the following, we show how the random oracle model is used to analyze the security of a cryptosystem. Because the ciphertext is much longer than the plaintext, the system we describe is not as efficient as methods such as OAEP (see [Section 9.2](#)). However, the present system is a good illustration of the use of the random oracle model.

Let f be a one-way one-to-one function that Bob knows how to invert. For example, $f(x) = x^e \pmod{n}$, where (e, n) is Bob's public RSA key. Let H be a hash function. To encrypt a message m , which is assumed to have the same bitlength as the output of H , Alice chooses a random integer $r \pmod{n}$ and lets the ciphertext be

$$(y_1, y_2) = (f(r), H(r) \oplus m).$$

When Bob receives (y_1, y_2) , he computes

$$r = f^{-1}(y_1), \quad m = H(r) \oplus y_2.$$

It is easy to see that this decryption produces the original message m .

Let's assume that the hash function is a random oracle. We'll show that if Alice can succeed with significantly better than 50% probability, then she can invert f with significantly better than zero probability. Therefore, if f is truly a one-way function, the cryptosystem has the ciphertext indistinguishability property. To test this property, Alice and Carla play the CI Game from [Section 4.5](#). Carla chooses two ciphertexts, m_0 and m_1 , and gives them to Alice. Alice randomly chooses $b = 0$ or 1 and encrypts m_b , yielding the ciphertext (y_1, y_2) . She gives (y_1, y_2) to Carla, who tries to guess whether $b = 0$ or $b = 1$. Suppose that

$$\text{Prob}(\text{Carla guesses correctly}) \geq \frac{1}{2} + \epsilon.$$

Let $L = \{r_1, r_2, \dots, r_\ell\}$ be the set of y for which Carla can compute $f^{-1}(y)$. If $y_1 \in L$, then Carla computes the value r such that $f(r) = y_1$. She then asks the random oracle for the value $H(r)$, computes $y_2 \oplus H(r)$, and obtains m_b . Therefore, when $y_1 \in L$, Carla guesses correctly.

If $r \notin L$, then Carla does not know the value of $H(r)$. Since H is a random oracle, the possible values of $H(r)$ are randomly and uniformly distributed among all possible outputs, so $H(m) \oplus m_b$ is the same as encrypting m_b with a one-time pad. As we saw in [Section 4.4](#), this means that y_2 gives Alice no information about whether it comes from m_1 or from m_2 . So if $r \notin L$, Alice has probability $1/2$ of guessing the correct plaintext.

Therefore

$$\begin{aligned} \frac{1}{2} + \epsilon &= \text{Prob(Alice guesses correctly)} \\ &= \frac{1}{2} \text{Prob}(r \notin L) + 1 \cdot \text{Prob}(r \in L) \\ &= \frac{1}{2}(1 - \text{Prob}(x \in L)) + 1 \cdot \text{Prob}(r \in L) \\ &= \frac{1}{2} + \frac{1}{2}\text{Prob}(x \in L). \end{aligned}$$

It follows that $\text{Prob}(x \in L) = 2\epsilon$.

If we assume that it is computationally feasible for Alice to find b with probability at most ϵ , then we conclude that it is computationally feasible for Alice to guess r correctly with probability $\frac{1}{2} + \epsilon$. Therefore, if the function f is one-way, then the cryptosystem has the ciphertext indistinguishability property.

Note that it was important in the argument to assume that the values of H are randomly and uniformly distributed. If this were not the case, so the hash function had some bias, then Alice might have some method for

guessing correctly with better than 50% probability when $r \notin L$. Therefore, the assumption that the hash function is a random oracle is important.

Of course, a good hash function is probably close to acting like a random oracle. In this case, the above argument shows that the cryptosystem with an actual hash function should be fairly resistant to Alice guessing correctly. However, it should be noted that Canetti, Goldreich, and Halevi [Canetti et al.] have constructed a cryptosystem that is secure in the random oracle model but is not secure for any concrete choice of hash function. Fortunately, this construction is not one that would be used in practice.

The above procedure of reducing the security of a system to the solvability of some fundamental problem, such as the non-invertibility of a one-way function, is common in proofs of security. For example, in [Section 10.5](#), we reduced certain questions for the ElGamal public key cryptosystem to the solvability of Diffie-Hellman problems.

[Section 12.2](#) shows that most hash functions do not behave as random oracles with respect to multicollisions. This indicates that some care is needed when applying the random oracle model.

The use of the random oracle model in analyzing a cryptosystem is somewhat controversial. However, many people feel that it gives some indication of the strength of the system. If a system is not secure in the random oracle model, then it surely is not safe in practice. The controversy arises when a system is proved secure in the random oracle model. What does this say about the security of actual implementations? Different cryptographers will give different answers. However, at present, there seems to be no better method of analyzing the security that works widely.

12.4 Using Hash Functions to Encrypt

Cryptographic hash functions are some of the most widely used cryptographic tools, perhaps second only to block ciphers. They find applications in many different areas of information security. Later, in Chapter 13, we shall see an application of hash functions to digital signatures, where the fact that they shrink the representation of data makes the operation of creating a digital signature more efficient. We now look at how they may be used to serve the role of a cipher by providing data confidentiality.

A cryptographic hash function takes an input of arbitrary length and provides a fixed-size output that appears random. In particular, if we have two distinct inputs, then their hashes should be different. Generally, their hashes are very different. This is a property that hash functions share with good ciphers and is a property that allows us to use a hash function to perform encryption.

Using a hash function to perform encryption is very similar to a stream cipher in which the output of a pseudorandom number generator is XORed with the plaintext. We saw such an example when we studied the output feedback mode (OFB) of a block cipher. Much like the block cipher did for OFB, the hash function creates a pseudorandom bit stream that is XORed with the plaintext to create a ciphertext.

In order to make a cryptographic hash function operate as a stream cipher, we need two components: a key shared between Alice and Bob, and an initialization vector. We shall soon address the issue of the initialization vector, but for now let us begin by assuming

that Alice and Bob have established a shared secret key K_{AB} .

Now, Alice could create a pseudorandom byte x_1 by taking the leftmost byte of the hash of K_{AB} ; that is, $x_1 = L_8(h(K_{AB}))$. She could then encrypt a byte of plaintext p_1 by XORing with the random byte x_1 to produce a byte of ciphertext

$$c_1 = p_1 \oplus x_1.$$

But if she has more than one byte of plaintext, then how should continue? We use feedback, much like we did in OFB mode. The next pseudorandom byte should be created by $x_2 = L_8(h(K_{AB} \parallel x_1))$. Then the next ciphertext byte can be created by

$$c_2 = p_2 \oplus x_2.$$

In general, the pseudorandom byte x_j is created by $x_j = L_8(h(K_{AB} \parallel x_{j-1}))$, and encryption is simply XORing x_j with the plaintext p_j . Decryption is a simple matter, as Bob must merely recreate the bytes x_j and XOR with the ciphertext c_j to get out the plaintext p_j .

There is a simple problem with this procedure for encryption and decryption. What if Alice wants to encrypt a message on Monday, and a different message on Wednesday? How should she create the pseudorandom bytes? If she starts all over, then the pseudorandom sequence x_j on Monday and Wednesday will be the same. This is not desirable.

Instead, we must introduce some randomness to make certain the two bit streams are different. Thus, each time Alice sends a message, she should choose a random initialization vector, which we denote by x_0 . She then starts by creating $x_1 = L_8(h(K_{AB} \parallel x_0))$ and proceeding as before. But now she must send x_0 to Bob, which she can do when she sends c_1 . If Eve intercepts x_0 , she is still not able to compute x_1 since she doesn't know

K_{AB} . In fact, if h is a good hash function, then x_0 should give no information about x_1 .

The idea of using a hash function to create an encryption procedure can be modified to create an encryption procedure that incorporates the plaintext, much in the same way as the CFB mode does.

12.5 Message Authentication Codes

When Alice sends a message to Bob, two important considerations are

1. Is the message really from Alice?
2. Has the message been changed during transmission?

Message authentication codes (MAC) solve these problems. Just as is done with digital signatures, Alice creates an appendix, the MAC, that is attached to the message. Thus, Alice sends (m, MAC) to Bob.

12.5.1 HMAC

One of the most commonly used is HMAC (= Hashed MAC), which was invented in 1996 by Mihir Bellare, Ran Canetti, and Hugo Krawczyk and is used in the IPSec and TLS protocols for secure authenticated communications.

To set up the protocol, we need a hash function H . For concreteness, assume that H processes messages in blocks of 512 bits. Alice and Bob need to share a secret key K . If K is shorter than 512 bits, append enough os to make its length be 512. If K is longer than 512 bits, it can be hashed to obtain a shorter key, so we assume that K has 512 bits.

We also form innerpadding and outerpadding strings:

```
opad = 5C5C5C ... 5C, ipad = 363636 ... 36,
```

which are binary strings of length 512, expressed in hexadecimal (5=0101, C=1100, etc.).

Let m be the message. Then Alice computes

$$HMAC(m, K) = H((K \oplus opad) \parallel H((K \oplus ipad) \parallel m)).$$

In other words, Alice first does the natural step of computing $H((K \oplus ipad) \parallel m)$. But, as pointed out in [Section 11.3](#), this is susceptible to length extension attacks for hash functions based on the Merkle-Damgård construction. So she prepends $K \oplus opad$ and hashes again. This seems to be resistant to all known attacks.

Alice now sends the message m , either encrypted or not, along with $HMAC(m, K)$, to Bob. Since Bob knows K , he can compute $HMAC(m, K)$. If it agrees with the value that Alice sent, he assumes that the message is from Alice and that it is the message that Alice sent.

If Eve tries to change m to m' , then $H(K \oplus ipad \parallel m')$ should differ from $H(K \oplus ipad \parallel m)$ (collision resistance) and therefore $HMAC(m', K)$ should differ from $HMAC(K, m)$ (again, by collision resistance). Therefore, $HMAC(m, K)$ tells Bob that the message is authentic. Also, it seems unlikely that Eve can authenticate a message m without knowing K , so Bob is sure that Alice sent the message.

For an analysis of the security of HMAC, see [Bellare et al.]

12.5.2 CBC-MAC

An alternative to using hash functions is to use a block cipher in CBC mode. Let $E_K()$ be a block cipher, such as AES, using a secret key K that is shared between Alice and Bob. We may create an appendix very similar to the

output of a keyed hash function by applying $E_K()$ in CBC mode to the entire message m and using the final output of the CBC operation as the MAC.

Suppose that our message m is of the form

$m = (m_1, m_2, \dots, m_l)$, where each m_j is a block of the message that has the same length as the block length of the encryption algorithm $E_K()$. The last block m_l may require padding in order to guarantee that it is a full block. Recall that the CBC encryption procedure is given by

$$C_j = E_K(m_j \oplus C_{j-1}) \quad j = 1, 2, \dots, l,$$

where $C_0 = IV$ is the initialization vector. The CBC-MAC is then given as

$$MAC = (IV, C_l)$$

and Alice then sends Bob (m, MAC) .

CBC-MAC has many of the same concerns as keyed hash functions. For example, CBC-MAC also suffers from its own form of length extension attacks. Let $MAC(a)$ correspond to the CBC-MAC of a message a . Suppose that Eve has found two messages a and b with

$MAC(a) = MAC(b)$. Then Eve knows that

$MAC(a, x) = MAC(b, x)$ for an additional block x .

If Eve can convince Alice to authenticate $a' = [a, x]$, she can swap a' with $b' = [b, x]$ to create a forged document that appears valid. Of course, the trick is in convincing Alice to authenticate a' , but it nonetheless is a concern with CBC-MAC.

When using CBC-MAC, one should also be careful not to use the key K for purposes other than authentication. A different key must be used for message confidentiality than for calculating the message authentication code. In particular, if one uses the same key for confidentiality as for authentication, then the output of CBC blocks during

the encryption of a message for confidentiality could be the basis for a forgery attack against CBC-MAC.

12.6 Password Protocols

We now look at how the communications take place in a password-based authentication protocol, as is commonly used to log onto computer systems. Password-based authentication is a popular approach to authentication because it is much easier for people to remember a phrase (the *password*) than it is to remember a long, cryptographic response.

In a password protocol, we have a user, Alice, and a verifier, Veronica. Veronica is often called a host and might, for example, be a computer terminal or a smartphone or an email service. Alice and Veronica share knowledge of the password, which is a long-term secret that typically belongs to a much smaller space of possibilities than cryptographic keys and thus passwords have small entropy (that is, much less randomness is in a password).

Veronica keeps a list of all users and their passwords $\{(ID, P_{ID})\}$. For many hosts this list might be small since only a few users might log into a particular machine, while in other cases the password list can be more substantial. For example, email services must maintain a very large list of users and their passwords. When Alice wants to log onto Veronica's service, she must contact Veronica and tell her who she is and give her password. Veronica, in turn, will check to see if Alice's password is legitimate.

A basic password protocol proceeds in several steps:

1. Alice → Veronica: "Hello, I am Alice"
2. Veronica → Alice: "Password?"
3. Alice → Veronica: P_{Alice}

4. Veronica: Examines password file $\{(ID, P_{ID})\}$ and verifies that the pair $(Alice, P_{Alice})$ belongs to the password file. Service is granted if the password is confirmed.

This protocol is very straightforward and, as it stands, has many flaws that should stand out. Perhaps one of the most obvious problems is that there is no mutual authentication between Alice and Veronica; that is, Alice does not know that she is actually communicating with the legitimate Veronica and, vice versa, Veronica does not actually have any proof that she is talking with Alice. While the purpose of the password exchange is to authenticate Alice, in truth there is no protection from message replay attacks, and thus anyone can pretend to be either Alice or Veronica.

Another glaring problem is the lack of confidentiality in the protocol. Any eavesdropper who witnesses the communication exchange learns Alice's password and thus can imitate Alice at a later time.

Lastly, another more subtle concern is the storage of the password file. The storage of $\{(ID, P_{ID})\}$ is a design liability as there are no guarantees that a system administrator will not read the password file and leak passwords. Similarly, there is no guarantee that Veronica's system will not be hacked and the password file stolen.

We would like the passwords to be protected while they are stored in the password file. Needham proposed that, instead of storing $\{(ID, P_{ID})\}$, Veronica should store $\{(ID, h(P_{ID}))\}$, where h is a one-way function that is difficult to invert, such as a cryptographic hash function. In this case, Step 4 involves Veronica checking $h(P_{Alice})$ against $\{(ID, h(P_{ID}))\}$. This basic scheme is what was used in the original Unix password system, where the function h was the variant of DES that used Salt (see [Chapter 7](#)). Now, an adversary who gets the file $\{(ID, h(P_{ID}))\}$ can't use this to respond in Step 3.

Nevertheless, although the revised protocol protects the password file, it does not address the eavesdropping concern. While we could attempt to address eavesdropping using encryption, such an approach would require an additional secret to be shared between Alice and Veronica (namely an encryption key). In the following, we present two solutions that do not require additional secret information to be shared between Alice and Veronica.

12.6.1 The Secure Remote Password protocol

Alice wants to log in to a server using her password. A common way of doing this is the Secure Remote Password protocol (SRP).

First, the system needs to be set up to recognize Alice and her password. Alice has her login name I and password P . Also, the server has chosen a large prime p such that $(p - 1)/2$ is also prime (this is to make the discrete logarithm problem mod p hard) and a primitive root α mod p . Finally, a cryptographic hash function H such as SHA256 or SHA3 is specified.

A random bitstring s is chosen (this is called “salt”), and $x = H(s \parallel I \parallel P)$ and $v \equiv \alpha^x \pmod{p}$ are computed. The server discards P and x , but stores s and v along with Alice’s identification I . Alice saves only I and P .

When Alice wants to log in, she and the server perform the following steps:

1. Alice sends I to the server and the server retrieves the corresponding s and v .
2. The server sends s to Alice, who computes $x = H(s \parallel I \parallel P)$.
3. Alice chooses a random integer $a \pmod{p-1}$ and computes $A \equiv \alpha^a \pmod{p}$. She sends A to the server.

4. The server chooses a random $b \bmod p - 1$, computes $B \equiv 3v + \alpha^b \bmod p$, and sends B to Alice.
5. Both Alice and the server compute $u = H(A \parallel B)$.
6. Alice computes $S \equiv (B - 3\alpha^x)^{a+ux} \pmod{p-1}$ and the server computes S as $(Av^u)^b$ (these yield the same S ; see below).
7. Alice computes $M_1 = H(A \parallel B \parallel S)$ and sends M_1 to the server, which checks that this agrees with the value of M_1 that is computed using the server's values of A, B, S .
8. The server computes $M_2 = H(A \parallel M_1 \parallel S)$ and sends M_2 to Alice. She checks that this agrees with the value of M_2 she computes with her values of A, M_1, S .
9. Both Alice and the server compute $K = H(S)$ and use this as the session key for communications.

Several comments are in order. We'll number them corresponding to the steps in the protocol.

1. The server does not directly store P or a hash of P . The hash of P is stored in a form protected by a discrete logarithm problem. Originally, $x = H(s \parallel P)$ was used and the salt s was included to slow down brute force attacks on passwords. Eve can try various values of P , trying to match a value of v , until someone's password is found (this is called a "dictionary attack"). If a sufficiently long bitstring s is included, this attack becomes infeasible. The current version of SRP includes I in the hash to get x , so Eve needs to attack an individual entry. Since Eve knows an individual's salt (if she obtained access to the password file), the salt is essentially part of the identification and does not slow down the attack on the individual's password.
2. Sending s to Alice means that Alice does not need to remember s .
3. This is essentially the start of a protocol similar to the Diffie-Hellman key exchange.
4. Earlier versions of SRP used $B \equiv v + \alpha^b$. But this meant that an attacker posing as the server could choose a random x' , compute $v' \equiv \alpha^{x'}$, and use $B \equiv \alpha^{x'} + \alpha^b$, thus allowing the attacker to check whether one of b, x' is the hash value x . In effect, this could speed up the attack by a factor of 2. The 3 is included to avoid this.
5. In an earlier version of SRP, u was chosen randomly by the server and sent to Alice. However, if the server sends u before A is received (for example, it might seem more efficient for the server to send both s and u in Step 2), there is a possible attack. See [Exercise 16](#). The present method of having $u = H(A \parallel B)$ ensures that u is determined after A .

6. Let's show that the two values of S are equal:

$$(B - 3\alpha^x)^{a+ux} \equiv (B - 3v)^{a+ux} \equiv (\alpha^b)^{a+ux} \equiv \alpha^{ab}\alpha^{uxb}$$

and

$$(Av^u)^b \equiv (\alpha^a(\alpha^x)^u)^b \equiv \alpha^{ab}\alpha^{uxb}.$$

Therefore, they agree. Note the hints of the Diffie-Hellman protocol, where α^{ab} is computed in two ways.

Since the value of B changes for each login, the value of S also changes, so an attacker cannot simply reuse some successful S .

7. Up to this point, the server has no assurance that the communications are with Alice. Anyone could have sent Alice's I and sent a random A . Alice and the server have computed S , but they don't know that their values agree. If they do, it is very likely that the correct x , hence the correct P , is being used. Checking M_1 shows that the values of S agree because of the collision resistance of the hash function. Of course, if the values of S don't agree, then Alice and the server will produce different session keys K in the last step, so communications will fail for this reason, too. But it seems better to terminate the protocol earlier if something is wrong.
8. How does Alice know that she is communicating with the server? This step tells Alice that the server's value of S matches hers, so it is very likely that the entity she is communicating with knows the correct x . Of course, someone who has hacked into the password file has all the information that the server has and can therefore masquerade as the server. But otherwise Alice is confident that she is communicating with the server.
9. At the point, Alice and the server are authenticated to each other. The session key serves as the secret key for communications between Alice and the server during the current session.

Observe that B , M_1 , and M_2 are the only numbers that are transmitted that depend on the password. The value of B contains $v \equiv \alpha^x$, but this is masked by adding on the random number α^b . The values of M_1 and M_2 contain S , which depends on x , but it is safely hidden inside a hash function. Therefore, if is very unlikely that someone who eavesdrops on communications between Alice and the server will obtain any useful information. For more on the security and design considerations, see [Wu1], [Wu2].

12.6.2 Lamport's protocol

Another method was proposed by Lamport. The protocol, which we now introduce, is an example of what is known as a *one-time* password scheme since each run of the protocol uses a temporary password that can only be used once. Lamport's one-time password protocol is a good example of a special construction using one-way (specifically, hash) functions that shows up in many different applications.

To start, we assume that Alice has a password P_{Alice} , that Veronica has chosen a large integer n , and that Alice and Veronica have agreed upon a one-way function h . A good choice for such an h is a cryptographic hash function, such as SHA256 or SHA3, which we described in [Chapter 11](#). Veronica calculates

$h^n(P_{Alice}) = h(h(\cdots(h(P_{Alice}))\cdots))$, and stores Alice's entry $(Alice, h^n(P_{Alice}), n)$ in a password file. Now, when Alice wants to authenticate herself to Veronica, she uses the revised protocol:

1. Alice → Veronica: "Hello, I am Alice."
2. Veronica → Alice: n , "Password?"
3. Alice → Veronica: $r = h^{n-1}(P_{Alice})$
4. Veronica takes r , and checks to see whether $h(r) = h^n(P_{Alice})$. If the check passes, then Veronica updates Alice's entry in the password as $(Alice, h^{n-1}(P_{Alice}), n - 1)$, and Veronica grants Alice access to her services.

At first glance, this protocol might seem confusing, and to understand how it works in practice it is useful to write out the following chain of hashes, known as a **hash chain**:

$$h^n(P_{Alice}) \xleftarrow{h} h^{n-1}(P_{Alice}) \xleftarrow{h} h^{n-2}(P_{Alice}) \xleftarrow{h} \cdots \xleftarrow{h} h(P_{Alice}) \xleftarrow{h} P_{Alice}.$$

For the first run of the protocol, in step 2, Veronica will tell Alice n and ask Alice the corresponding password

that will hash to $h^n(P_{Alice})$. In order for Alice to correctly respond, she must calculate $h^{n-1}(P_{Alice})$, which she can do since she has the original password. After Alice is successfully verified, Veronica will throw away $h^n(P_{Alice})$ and update the password file to contain $(Alice, h^{n-1}(P_{Alice}), n - 1)$.

Now suppose that Eve saw $h^{n-1}(P_{Alice})$. This won't help her because the next time the protocol is run, in step 2 Veronica will issue $n - 1$ and thereby ask Alice for the corresponding password that will hash to $h^{n-1}(P_{Alice})$. Although Eve has $h^{n-1}(P_{Alice})$, she cannot determine the required response $r = h^{n-2}(P_{Alice})$.

The protocol continues to run, with Veronica updating her password file until she runs to the end of the hash chain. At that point, Alice and Veronica must renew the password file by changing the initial password P_{Alice} to a new password.

This protocol that we have just examined is the basis for the S/Key protocol, which was implemented in Unix operating systems in the 1980s. Although the S/Key protocol's use of one-time passwords achieves its purpose of protecting the password exchange from eavesdropping, it is nevertheless weak when one considers an active adversary. In particular, the fact that the counter n in step 2 is sent in the clear, and the lack of authentication, is the basis for an intruder-in-the-middle attack.

In the intruder-in-the-middle attack described below, Alice intends to communicate with Veronica, but the active adversary Malice intercepts the communications between Alice and Veronica and sends her own.

1. Alice → Malice (Veronica): "Hello, I am Alice."
2. Malice → Veronica: "Hello, I am Alice."
3. Veronica → Malice: n , "Password?"

4. Malice \rightarrow Alice: $n - 1$, “Password?”
5. Alice \rightarrow Malice: $r_1 = h^{n-2}(P_{Alice})$
6. Malice \rightarrow Veronica: $r_2 = h^{n-1}(P_{Alice}) = h(h^{n-2}(P_{Alice}))$.
7. Veronica takes r_2 , and checks to see whether $h(r_2) = h^n(P_{Alice})$.
The check will pass, and Veronica will think she is communicating with Alice, when really she is corresponding with Malice.

One of the problems with the protocol is that there is no authentication of the origin of the messages. Veronica does not know whether she is really communicating with Alice, and likewise Alice does not have a strong guarantee that she is communicating with Veronica.

The lack of origin authentication also provides the means to launch another clever attack, known as the small value attack. In this attack, Malice impersonates Veronica and asks Alice to respond to a small n . Then Malice intercepts Alice’s answer and uses that to calculate the rest of the hash chain. For example, if Malice sent $n = 10$, she would be able to calculate $h^{10}(P_{Alice})$, $h^{11}(P_{Alice})$, $h^{12}(P_{Alice})$, and so on. The small value of n allows Malice to hijack the majority of the hash chain, and thereby imitate Alice at a later time.

As a final comment, we note that the protocol we described is actually different than what was originally presented by Lamport. In Lamport’s original one-time password protocol, he required that Alice and Veronica keep track of n on their own without exchanging n . This has the benefit of protecting the protocol from active attacks by Malice where she attempts to use n to her advantage. Unfortunately, Lamport’s scheme required that Alice and Veronica stayed synchronized with each other, which in practice turns out to be difficult to ensure.

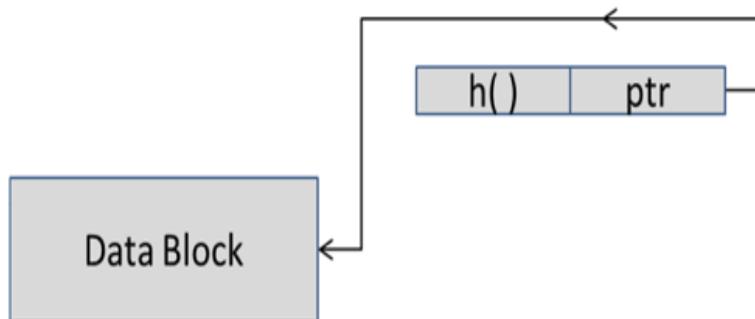
12.7 Blockchains

One variation of the concept of a hash chain is the blockchain. Blockchains are a technology that has garnered a lot of attention since they provide a convenient way to keep track of information in a secure and distributed manner.

Hash chains are iterations of hash functions, while blockchains are hash chains that have extra structure. In order to understand blockchains, we need to look at several different building blocks.

Let us start with hash pointers. A **hash pointer** is simply a pointer to where some data is stored, combined with a cryptographic hash of the value of the data that is being pointed at. We may visualize a hash pointer as something like Figure 12.1.

Figure 12.1 A hash pointer consists of a pointer that references a block of data, and a cryptographic hash of that data



The hash pointer is useful in that it gives a means to detect alterations of the block of data. If someone alters the block, then the hash contained in the hash pointer will not match the hash of the altered data block, assuming of course that no one has altered the hash pointer.

If we want to make it harder for someone to alter the hash, then we can create an ordered collection of data blocks, each with a hash pointer to a previous block. This is precisely the idea behind a blockchain. A blockchain is a collection of data blocks and hash pointers arranged in a data structure known as a linked list. Normal linked lists involve series of blocks of data that are each paired with a pointer to a previous block of data and its pointer. Blockchains, however, replace the pointer in a linked list with a hash pointer, as depicted in Figure 12.2.

Figure 12.2 A blockchain consists of a collection of blocks. Each block contains a data block, a hash of the previous block, and a pointer to the previous block. A final hash pointer references the end of the blockchain

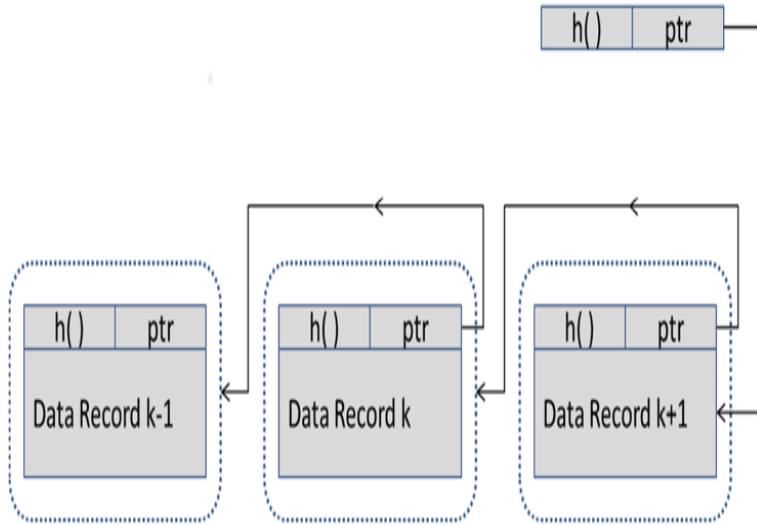


Figure 12.2 Full Alternative Text

Blockchains are useful because they allow entities to create a ledger of data that can be continually updated by one or more users. An initial version of the ledger is created, and then subsequent updates to the ledger reference previous updates to the ledger. In order to accomplish this, the basic structure of a blockchain consists of three main components: data, which is the ledger update; a pointer that tells where the previous block is located; and a digest (hash) that allows one to verify that the entire contents of the previous block have not changed. Suppose that a large record of blocks has been stored in a blockchain, such as in [Figure 12.2](#), with the final hash value not paired with any data. What happens, then, if an adversary comes along and wants to modify the data in block $k - 1$? If the data in block $k - 1$ is altered, then the hash contained in block k will not match the hash of the modified data in block $k - 1$. This forces the adversary to have to modify the hash in block k . But the hash of block $k + 1$ was calculated based on the entire block k (i.e. including the k -th hash), and therefore there will now be a mismatch between the hash of block k and the hash pointer in block $k + 1$. This process forces the adversary to continue modifying blocks until the end of the blockchain is reached. This requires a significant effort on the part of the adversary,

but is possible since the adversary can calculate the hash values that he needs to replace with. Ultimately, in order to prevent the adversary from succeeding, we require that the final hash value is stored in a manner that prevents the adversary from modifying it, thereby providing a final form of evidence preventing modification to the blockchain.

In practice, the data contained in each block can be quite large, consisting of many data records, and therefore one will often find another hash-based data structure used in blockchains. This data structure uses hash pointers arranged in a binary tree, and is known as a **Merkle tree**. Merkle trees are useful as they make it easy for someone to prove that a certain piece of data exists within a particular block of data.

In a Merkle tree, one has a collection of n records of data that have been arranged as the leaves of a binary tree, such as shown in Figure 12.3. These data records are then grouped in pairs of two, and for each pair two hash pointers are created, one pointing to the left data record and another to the right data record. The collection of hash pointers then serve as the data record in the next level of the tree, which are subsequently grouped in pairs of two. Again, for each pair two hash pointers are created, one pointing to the left record and the other to the right data record. We proceed up the tree until we reach the end, which is a single record that corresponds to the tree's root. The hash of this record is stored, giving one the ability to make certain that the entire collection of data records contained within the Merkle tree has not been altered.

Figure 12.3 A Merkle tree consists of data records that

have been arranged in pairs. Pairs of hash pointers point to the data records below them, and serve as a data record for the next level up in the binary tree. A final hash pointer references the head of the tree

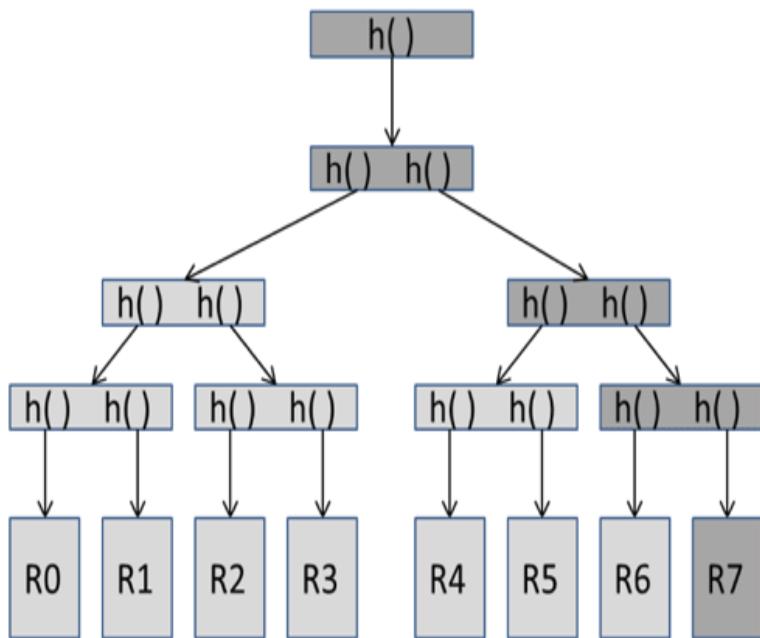


Figure 12.3 Full Alternative Text

Now suppose that someone wants to prove to you that a specific data record exists within a block. To do this, all that they need to show you is the data record, along with the hashes on the path from the data record to the root of the Merkle tree. In particular, one does not need to show all of the other data records, one only needs to show the hashes at the higher levels. This process is efficient, requiring roughly $\log(n)$ items from the Merkle tree to be shown, where n is the number of blocks of data

recorded. For example, to verify that $R7$ is in Figure 12.3, someone needs to show you $R7, h(R7), h(R6)$ (but not $R6$), $h(h(R6), h(R7))$, the two inputs and the hash at the next level up, and the two inputs and the hash at the top. You can check these hash computations, and, since the hash function is collision-resistant, you can be sure that $R7$ is there and has not been changed. Otherwise, at some level, a hash value has been changed and a collision has been found.

12.8 Exercises

1. In a family of four, what is the probability that no two people have birthdays in the same month? (Assume that all months have equal probabilities.)
2. Each person in the world flips 100 coins and obtains a sequence of length 100 consisting of Heads and Tails. (There are $2^{100} \approx 10^{30}$ possible sequences.) Assume that there are approximately 10^{10} people in the world. What is the probability that two people obtain the same sequence of Heads and Tails? Your answer should be accurate to at least two decimal places.
3.
 1. Let E_K be an encryption function with N possible keys K , N possible plaintexts, and N possible ciphertexts. Assume that if you know the encryption key K , then it is easy to find the decryption function D_K (therefore, this problem does not apply to public key methods). Suppose that, for each pair (K_1, K_2) of keys, it is possible to find a key K_3 such that $E_{K_1}(E_{K_2}(m)) = E_{K_3}(m)$ for all plaintexts m . Assume also that for every plaintext–ciphertext pair (m, c) , there is usually only one key K such that $E_K(m) = c$. Suppose that you know a plaintext–ciphertext pair (m, c) . Give a birthday attack that usually finds the key K in approximately $2\sqrt{N}$ steps. (Remark: This is much faster than brute force searching through all keys K , which takes time proportional to N .)
 2. Show that the shift cipher (see [Section 2.1](#)) satisfies the conditions of part (a), and explain how to attack the shift cipher mod 26 using two lists of length 6. (Of course, you could also find the key by simply subtracting the plaintext from the ciphertext; therefore, the point of this part of the problem is to illustrate part (a).)
4. Alice uses double encryption with a block cipher to send messages to Bob, so $c = E_{K_1}(E_{K_2}(m))$ gives the encryption. Eve obtains a plaintext–ciphertext pair (m, c) and wants to find K_1, K_2 by the Birthday Attack. Suppose that the output of E_K has N bits. Eve computes two lists:
 1. $E_K(m)$ for $3 \cdot 2^{N/2}$ randomly chosen keys K .
 2. $D_L(c)$ for $3 \cdot 2^{N/2}$ randomly chosen keys L .

1. Why is there a very good chance that Eve finds a key pair (L, K) such that $c = E_L(E_K(m))$?
2. Why is it unlikely that (L, K) is the correct key pair? (Hint: Look at the analysis of the Meet-in-the-Middle Attack in Section 6.5.)
3. What is the difference between the Meet-in-the-Middle Attack and what Eve does in this problem?

5. Each person who has ever lived on earth receives a deck of 52 cards and thoroughly shuffles it. What is the probability that two people have the cards in the same order? It is estimated that around 1.08×10^{11} people have ever lived on earth. The number of shuffles of 52 cards is $52! \approx 8 \times 10^{67}$.

6. Let p be a 300-digit prime. Alice chooses a secret integer k and encrypts messages by the function $E_k(m) = m^k \pmod{p}$.
 1. Suppose Eve knows a cipher text c and knows the prime p . She captures Alice's encryption machine and decides to try to find m by a birthday attack. She makes two lists. The first list contains $c \cdot E_k(x)^{-1} \pmod{p}$ for some random choices of x . Describe how to generate the second list, state approximately how long the two lists should be, and describe how Eve finds m if her attack is successful.
 2. Is this attack practical? Why or why not?

7. There are approximately 3×10^{147} primes with 150 digits. There are approximately 10^{85} particles in the universe. If each particle chooses a random 150-digit prime, do you think two particles will choose the same prime? Explain why or why not.

8. If there are five people in a room, what is the probability that no two of them have birthdays in the same month? (Assume that each person has probability $1/12$ of being born in any given month.)

9. You use a random number generator to generate 10^9 random 15-digit numbers. What is the probability that two of the numbers are equal? Your answer should be accurate enough to say whether it is likely or unlikely that two of the numbers are equal.

10. Nelson has a hash function H_1 that gives an output of 60 bits. Friends tell him that this is not a big enough output, so he takes a strong hash function H_2 with a 200-bit output and uses $H(x) = H_2(H_1(x))$ as his hash function. That is, he first hashes with his old hash function, then hashes the result with the strong hash function to get a 200-bit output, which he thinks is much better. The new hash function H can be computed quickly. Does it

have preimage resistance, and does it have strong collision resistance? Explain your answers. (Note: Assume that computers can do up to $2^{50} \approx 10^{15}$ computations for this problem. Also, since it is essentially impossible to prove rigorously that most hash functions have preimage resistance or collision resistance, if your answer to either of these is “yes” then your explanation is really an explanation of why it is probably true.)

11. Bob signs contracts by signing the hash values of the contracts. He is using a hash function H with a 50-bit output. Eve has a document M that states that Bob will pay her a lot of money. Eve finds a file with 10^9 documents that Bob has signed. Explain how Eve can forge Bob's signature on a document (closely related to M) that states that Bob will pay Eve a lot of money. (Note: You may assume that Eve can do up to 2^{30} calculations.)

12. This problem derives the formula (12.1) for the probability of at least one match in a list of length r when there are N possible birthdays.

1. Let $f(x) = \ln(1-x) + x$ and $g(x) = \ln(1-x) + x + x^2$. Show that $f'(x) \leq 0$ and $g'(x) \geq 0$ for $0 \leq x \leq 1/2$.

2. Using the facts that $f(0) = g(0) = 0$ and f is decreasing and g is increasing, show that

$$-x - x^2 \leq \ln(1-x) \leq -x \quad \text{for } 0 \leq x \leq 1/2.$$

3. Show that if $r \leq N/2$, then

$$-\frac{(r-1)r}{2N} - \frac{r^3}{3N^2} \leq \sum_{j=1}^{r-1} \ln\left(1 - \frac{j}{N}\right) \leq -\frac{(r-1)r}{2N}.$$

(Hint: $\sum_{j=1}^{r-1} j = (r-1)r/2$ and $\sum_{j=1}^{r-1} j^2 = (r-1)r(2r-1)/6 < r^3/3$.)

4. Let $\lambda = r^2/(2N)$ and assume that $\lambda \leq N/8$ (this implies that $r \leq N/2$). Show that

$$e^{-\lambda} e^{c_1/\sqrt{N}} \leq \prod_{j=1}^{r-1} \left(1 - \frac{j}{N}\right) \leq e^{-\lambda} e^{c_2/\sqrt{N}},$$

with $c_1 = \sqrt{\lambda/2} - \frac{1}{3}(2\lambda)^{3/2}$ and $c_2 = \sqrt{\lambda/2}$.

5. Observe that when N is large, $e^{c/\sqrt{N}}$ is close to 1. Use this to show that as N becomes large and λ is constant with $\lambda \leq N/8$, then we have the approximation

$$\prod_{j=1}^{r-1} \left(1 - \frac{j}{N}\right) \approx e^{-\lambda}.$$

13. Suppose $f(x)$ is a function with n -bit outputs and with inputs much larger than n bits (this implies that collisions must exist). We know that, with a birthday attack, we have probability $1/2$ of finding a collision in approximately $2^{n/2}$ steps.

1. Suppose we repeat the birthday attack until we find a collision. Show that the expected number of repetitions is

$$1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{16} + \dots = 2$$

(one way to evaluate the sum, call it S , is to write

$$S - \frac{1}{2}S = \frac{1}{2} + (2-1)\frac{1}{4} + (3-2)\frac{1}{8} + \dots = 1.$$

2. Assume that each evaluation of f takes time a constant times n . (This is realistic since the inputs needed to find collisions can be taken to have $2n$ bits, for example.) Show that the expected time to find a collision for the function f is a constant times $n 2^{n/2}$.
3. Show that the expected time to produce the messages $m_0, m'_0, \dots, m_{t-1}, m'_{t-1}$ in [Section 12.2](#) is a constant times $tn 2^{n/2}$.

14. Suppose we have an iterative hash function, as in [Section 11.3](#), but suppose we adjust the function slightly at each iteration. For concreteness, assume that the algorithm proceeds as follows. There is a compression function f that operates on inputs of a fixed length. There is also a function g that yields outputs of a fixed length, and there is a fixed initial value IV . The message is padded to obtain the desired format, then the following steps are performed:

1. Split the message M into blocks M_1, M_2, \dots, M_ℓ .
2. Let H_0 be the initial value IV .
3. For $i = 1, 2, \dots, \ell$, let $H_i = f(H_{i-1}, M_i \parallel g(i))$.
4. Let $H(M) = H_\ell$.

Show that the method of [Section 12.2](#) can be used to produce multicollisions for this hash function.

15. Some of the steps of SRP are similar to the Diffie-Hellman key exchange. Why not use Diffie-Hellman to log in, using the following protocol? Alice and the server use Diffie-Hellman to establish a key K . Or they could use a public key method to transmit a secret key K from the server to Alice. Then they use K , along with a symmetric system such as AES, to submit Alice's password P . Finally, the hash of the password is compared to what is stored in the computer's password file.

1. Show how Eve can do an intruder-in-the-middle attack and obtain Alice's password.
 2. In order to avoid the attack in part (a), Alice and the server decide that Alice should send the hash of her password. Show that if Eve uses an intruder-in-the-middle attack, then she can log in to the server, pretending to be Alice.
 3. Alice and the server have another idea. The server sends Alice a random r and Alice sends $H(r \parallel P)$ to the server. Show how Eve can use an intruder-in-the-middle-attack to log in as Alice.
16. Suppose that in SRP, the number u is chosen randomly by the server and sent to Alice at the same time that s is sent. Suppose Eve has obtained v from the server's password file. Eve chooses a random a , computes $A \equiv g^a v^{-u} \pmod{p}$, and sends this value of A to the server. Then Eve computes S as $(B - 3v)^a \pmod{p}$. Show that these computations appear to be valid to the server, so Eve can log in as Alice.

12.9 Computer Problems

1.
 1. If there are 30 people in a classroom, what is the probability that at least two have the same birthday? Compare this to the approximation given by formula (8.1).
 2. How many people should there be in a classroom in order to have a 99% chance that at least two have the same birthday? (Hint: Use the approximation to obtain an approximate answer. Then use the product, for various numbers of people, until you find the exact answer.)
 3. How many people should there be in a classroom in order to have 100% probability that at least two have the same birthday?
2. A professor posts the grades for a class using the last four digits of the Social Security number of each student. In a class of 200 students, what is the probability that at least two students have the same four digits?

Chapter 13 Digital Signatures

For years, people have been using various types of signatures to associate their identities to documents. In the Middle Ages, a nobleman sealed a document with a wax imprint of his insignia. The assumption was that the noble was the only person able to reproduce the insignia. In modern transactions, credit card slips are signed. The salesperson is supposed to verify the signature by comparing with the signature on the card. With the development of electronic commerce and electronic documents, these methods no longer suffice.

For example, suppose you want to sign an electronic document. Why can't you simply digitize your signature and append it to the document? Anyone who has access to it can simply remove the signature and add it to something else, for example, a check for a large amount of money. With classical signatures, this would require cutting the signature off the document, or photocopying it, and pasting it on the check. This would rarely pass for an acceptable signature. However, such an electronic forgery is quite easy and cannot be distinguished from the original.

Therefore, we require that digital signatures cannot be separated from the message and attached to another. That is, the signature is not only tied to the signer but also to the message that is being signed. Also, the digital signature needs to be easily verified by other parties. Digital signature schemes therefore consist of two distinct steps: the signing process, and the verification process.

In the following, we first present two signature schemes. We also discuss the important “birthday attacks” on

signature schemes.

Note that we are not trying to encrypt the message m . In fact, often the message is a legal document, and therefore should be kept public. However, if necessary, a signed message may be encrypted after it is signed. (This is done in PGP, for example. See [Section 15.6](#).)

13.1 RSA Signatures

Bob has a document m that Alice agrees to sign. They do the following:

1. Alice generates two large primes p, q , and computes $n = pq$. She chooses e_A such that $1 < e_A < \phi(n)$ with $\gcd(e_A, \phi(n)) = 1$, and calculates d_A such that $e_A d_A \equiv 1 \pmod{\phi(n)}$. Alice publishes (e_A, n) and keeps private d_A, p, q .
2. Alice's signature is
$$s \equiv m^{d_A} \pmod{n}.$$
3. The pair (m, s) is then made public.

Bob can then verify that Alice really signed the message by doing the following:

1. Download Alice's (e_A, n) .
2. Calculate $z \equiv s^{e_A} \pmod{n}$. If $z = m$, then Bob accepts the signature as valid; otherwise the signature is not valid.

Suppose Eve wants to attach Alice's signature to another message m_1 . She cannot simply use the pair (m_1, s) , since $s^{e_A} \not\equiv m_1 \pmod{n}$. Therefore, she needs s_1 with $s_1^{e_A} \equiv m_1 \pmod{n}$. This is the same problem as decrypting an RSA "ciphertext" m_1 to obtain the "plaintext" s_1 . This is believed to be hard to do.

Another possibility is that Eve chooses s_1 first, then lets the message be $m_1 \equiv s_1^{e_A} \pmod{n}$. It does not appear that Alice can deny having signed the message m_1 under the present scheme. However, it is very unlikely that m_1 will be a meaningful message. It will probably be a random sequence of characters, and not a message committing her to give Eve millions of dollars. Therefore, Alice's claim that it has been forged will be believable.

There is a variation on this procedure that allows Alice to sign a document without knowing its contents. Suppose Bob has made an important discovery. He wants to record publicly what he has done (so he will have priority when it comes time to award Nobel prizes), but he does not want anyone else to know the details (so he can make a lot of money from his invention). Bob and Alice do the following. The message to be signed is m .

1. Alice chooses an RSA modulus n ($n = pq$, the product of two large primes), an encryption exponent e , and decryption exponent d . She makes n and e public while keeping p , q , d private. In fact, she can erase p , q , d from her computer's memory at the end of the signing procedure.
2. Bob chooses a random integer $k \pmod n$ with $\gcd(k, n) = 1$ and computes $t \equiv k^e m \pmod n$. He sends t to Alice.
3. Alice signs t by computing $s \equiv t^d \pmod n$. She returns s to Bob.
4. Bob computes $s/k \pmod n$. This is the signed message m^d .

Let's show that s/k is the signed message: Note that $k^{ed} \equiv (k^e)^d \equiv k \pmod n$, since this is simply the encryption, then decryption, of k in the RSA scheme. Therefore,

$$s/k \equiv t^d/k \equiv k^{ed}m^d/k \equiv m^d \pmod n,$$

which is the signed message.

The choice of k is random, so $k^e \pmod n$ is the RSA encryption of a random number, and hence random. Therefore, $k^e m \pmod n$ gives essentially no information about m (however, it would not hide a message such as $m = 0$). In this way, Alice knows nothing about the message she is signing.

Once the signing procedure is finished, Bob has the same signed message as he would have obtained via the standard signing procedure.

There are several potential dangers with this protocol. For example, Bob could have Alice sign a promise to pay him a million dollars. Safeguards are needed to prevent such problems. We will not discuss these here.

Schemes such as these, called **blind signatures**, have been developed by David Chaum, who has several patents on them.

13.2 The ElGamal Signature Scheme

The ElGamal encryption method from [Section 10.5](#) can be modified to give a signature scheme. One feature that is different from RSA is that, with the ElGamal method, there are many different signatures that are valid for a given message.

Suppose Alice wants to sign a message. To start, she chooses a large prime p and a primitive root α . Alice next chooses a secret integer a such that $1 \leq a \leq p - 2$ and calculates $\beta \equiv \alpha^a \pmod{p}$. The values of p , α , and β are made public. The security of the system will be in the fact that a is kept private. It is difficult for an adversary to determine a from (p, α, β) since the discrete log problem is considered difficult.

In order for Alice to sign a message m , she does the following:

1. Selects a secret random k with $1 \leq k \leq p - 2$ such that $\gcd(k, p - 1) = 1$.
2. Computes $r \equiv \alpha^k \pmod{p}$ (with $0 < r < p$).
3. Computes $s \equiv k^{-1}(m - ar) \pmod{p - 1}$.

The signed message is the triple (m, r, s) .

Bob can verify the signature as follows:

1. Download Alice's public key (p, α, β) .
2. Compute $v_1 \equiv \beta^r r^s \pmod{p}$, and $v_2 \equiv \alpha^m \pmod{p}$.
3. The signature is declared valid if and only if $v_1 \equiv v_2 \pmod{p}$.