

*"... the best introduction
to cryptography I've
ever seen.... The book
the National Security
Agency wanted never
to be published...."*

—Wired Magazine

**SECOND
EDITION**

APPLIED CRYPTOGRAPHY



**Protocols, Algorithms,
and Source Code in C**

BRUCE SCHNEIER

NF-1009
U.S. Patent 10,726,102

LOR, GORDON

from reviews of the first edition of

APPLIED CRYPTOGRAPHY

Protocols, Algorithms, and Source Code in C

"... the definitive text on the subject. . . ."

—*Software Development Magazine*

"... good reading for anyone interested in cryptography."

—*BYTE*

"This book should be on the shelf of any computer professional involved in the use or implementation of cryptography."

—*IEEE Software*

"... dazzling . . . fascinating. . . . This book *absolutely must* be on your bookshelf . . ."

—*PC Techniques*

"... comprehensive . . . an encyclopedic work . . ."

—*The Cryptogram*

"... a fantastic book on cryptography today. It belongs in the library of anyone interested in cryptography or anyone who deals with information security and cryptographic systems."

—*Computers & Security*

"An encyclopedic survey . . . could well have been subtitled 'The Joy of Encrypting' . . . a useful addition to the library of any active or would-be security practitioner."

—*Cryptologia*

"... encyclopedic . . . readable . . . well-informed . . . picks up where Dorothy Denning's classic *Cryptography and Data Security* left off a dozen years ago. . . . This book would be a bargain at twice the price."

—*login:*

"This is a marvelous resource—the best book on cryptography and its application available today."

—Dorothy Denning
Georgetown University

"... Schneier's book is an indispensable reference and resource. . . . I recommend it highly."

—Martin Hellman
Stanford University

Errata

A list of the errors found in this book along with corresponding corrections is updated periodically. For the most recent electronic version, send email to:

`schneier@counterpane.com`

For the most recent printed version, send a stamped, self-addressed envelope to:

AC Corrections
Counterpane Systems
7115 W. North Ave., Suite 16
Oak Park, IL 60302-1002

Readers are encouraged to distribute electronic or printed versions of this list to other readers of this book.

APPLIED CRYPTOGRAPHY, SECOND EDITION

PROTOCOLS, ALGORITHMS, AND SOURCE CODE IN C

BRUCE SCHNEIER



John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Katherine Schowalter
Editor: Phil Sutherland
Assistant Editor: Allison Roarty
Managing Editor: Robert Aronds
Text Design & Composition: North Market Street Graphics

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc. is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This text is printed on acid-free paper.

Copyright © 1996 by Bruce Schneier
Published by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought.

In no event will the publisher or author be liable for any consequential, incidental, or indirect damages (including damages for loss of business profits, business interruption, loss of business information, and the like) arising from the use or inability to use the protocols and algorithms in this book, even if the publisher or author has been advised of the possibility of such damages.

Some of the protocols and algorithms in this book are protected by patents and copyrights. It is the responsibility of the reader to obtain all necessary patent and copyright licenses before implementing in software any protocol or algorithm in this book. This book does not contain an exhaustive list of all applicable patents and copyrights.

Some of the protocols and algorithms in this book are regulated under the United States Department of State International Traffic in Arms Regulations. It is the responsibility of the reader to obtain all necessary export licenses before implementing in software for export any protocol or algorithm in this book.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging-in-Publication Data:

Schneier, Bruce

Applied Cryptography Second Edition : protocols, algorithms, and source code in C
/ Bruce Schneier.

p. cm.

Includes bibliographical references [p. 675].

ISBN 0-471-12845-7 (cloth : acid-free paper). — ISBN

0-471-11709-9 (paper : acid-free paper)

1. Computer security.
2. Telecommunication—Security measures.
3. Cryptography. I. Title.

QA76.9.A25S35 1996

005.8'2—dc20

95-12398
CIP

Printed in the United States of America
10 9 8 7 6 5 4

Contents in Brief

Foreword by Whitfield Diffie

Preface

About the Author

1 Foundations

Part I Cryptographic Protocols

2 Protocol Building Blocks

3 Basic Protocols

4 Intermediate Protocols

5 Advanced Protocols

6 Esoteric Protocols

Part II Cryptographic Techniques

7 Key Length

8 Key Management

9 Algorithm Types and Modes

10 Using Algorithms

Part III Cryptographic Algorithms

11 Mathematical Background

12 Data Encryption Standard (DES)

13 Other Block Ciphers

14 Still Other Block Ciphers

15 Combining Block Ciphers

16 Pseudo-Random-Sequence Generators and Stream Ciphers

17 Other Stream Ciphers and Real Random-Sequence Generators

18 One-Way Hash Functions

19 Public-Key Algorithms

20 Public-Key Digital Signature Algorithms

21 Identification Schemes

22 Key-Exchange Algorithms

23 Special Algorithms for Protocols

Part IV The Real World

24 Example Implementations

25 Politics

Afterword by Matt Blaze

Part V Source Code

References

Contents

Foreword by Whitfield Diffie xv

Preface xix

HOW TO READ THIS BOOK xx

ACKNOWLEDGMENTS xxii

About the Author xxiii

1 FOUNDATIONS 1

- 1.1 TERMINOLOGY 1
- 1.2 STEGANOGRAPHY 9
- 1.3 SUBSTITUTION CIPHERS AND TRANSPOSITION CIPHERS 10
- 1.4 SIMPLE XOR 13
- 1.5 ONE-TIME PADS 15
- 1.6 COMPUTER ALGORITHMS 17
- 1.7 LARGE NUMBERS 17

PART I CRYPTOGRAPHIC PROTOCOLS

2 PROTOCOL BUILDING BLOCKS 21

- 2.1 INTRODUCTION TO PROTOCOLS 21
- 2.2 COMMUNICATIONS USING SYMMETRIC CRYPTOGRAPHY 28
- 2.3 ONE-WAY FUNCTIONS 29
- 2.4 ONE-WAY HASH FUNCTIONS 30
- 2.5 COMMUNICATIONS USING PUBLIC-KEY CRYPTOGRAPHY 31
- 2.6 DIGITAL SIGNATURES 34
- 2.7 DIGITAL SIGNATURES WITH ENCRYPTION 41
- 2.8 RANDOM AND PSEUDO-RANDOM-SEQUENCE GENERATION 44

3 BASIC PROTOCOLS 47

- 3.1 KEY EXCHANGE 47
- 3.2 AUTHENTICATION 52
- 3.3 AUTHENTICATION AND KEY EXCHANGE 56
- 3.4 FORMAL ANALYSIS OF AUTHENTICATION AND KEY-EXCHANGE PROTOCOLS 65
- 3.5 MULTIPLE-KEY PUBLIC-KEY CRYPTOGRAPHY 68
- 3.6 SECRET SPLITTING 70
- 3.7 SECRET SHARING 71
- 3.8 CRYPTOGRAPHIC PROTECTION OF DATABASES 73

4 INTERMEDIATE PROTOCOLS 75

- 4.1 TIMESTAMPING SERVICES 75
- 4.2 SUBLIMINAL CHANNEL 79
- 4.3 UNDENIABLE DIGITAL SIGNATURES 81
- 4.4 DESIGNATED CONFIRMER SIGNATURES 82
- 4.5 PROXY SIGNATURES 83
- 4.6 GROUP SIGNATURES 84
- 4.7 FAIL-STOP DIGITAL SIGNATURES 85
- 4.8 COMPUTING WITH ENCRYPTED DATA 85
- 4.9 BIT COMMITMENT 86
- 4.10 FAIR COIN FLIPS 89
- 4.11 MENTAL POKER 92
- 4.12 ONE-WAY ACCUMULATORS 95
- 4.13 ALL-OR-NOTHING DISCLOSURE OF SECRETS 96
- 4.14 KEY ESCROW 97

5 ADVANCED PROTOCOLS 101

- 5.1 ZERO-KNOWLEDGE PROOFS 101
- 5.2 ZERO-KNOWLEDGE PROOFS OF IDENTITY 109
- 5.3 BLIND SIGNATURES 112
- 5.4 IDENTITY-BASED PUBLIC-KEY CRYPTOGRAPHY 115
- 5.5 OBLIVIOUS TRANSFER 116
- 5.6 OBLIVIOUS SIGNATURES 117
- 5.7 SIMULTANEOUS CONTRACT SIGNING 118
- 5.8 DIGITAL CERTIFIED MAIL 122
- 5.9 SIMULTANEOUS EXCHANGE OF SECRETS 123

6 ESOTERIC PROTOCOLS 125

- 6.1 SECURE ELECTIONS 125
- 6.2 SECURE MULTIPARTY COMPUTATION 134
- 6.3 ANONYMOUS MESSAGE BROADCAST 137
- 6.4 DIGITAL CASH 139

PART II CRYPTOGRAPHIC TECHNIQUES**7 KEY LENGTH 151**

- 7.1 SYMMETRIC KEY LENGTH 151
- 7.2 PUBLIC-KEY KEY LENGTH 158
- 7.3 COMPARING SYMMETRIC AND PUBLIC-KEY KEY LENGTH 165
- 7.4 BIRTHDAY ATTACKS AGAINST ONE-WAY HASH FUNCTIONS 165
- 7.5 HOW LONG SHOULD A KEY BE? 166
- 7.6 CAVEAT EMPTOR 168

8 KEY MANAGEMENT 169

- 8.1 GENERATING KEYS 170
- 8.2 NONLINEAR KEYSPACES 175
- 8.3 TRANSFERRING KEYS 176
- 8.4 VERIFYING KEYS 178
- 8.5 USING KEYS 179
- 8.6 UPDATING KEYS 180
- 8.7 STORING KEYS 180
- 8.8 BACKUP KEYS 181
- 8.9 COMPROMISED KEYS 182
- 8.10 LIFETIME OF KEYS 183
- 8.11 DESTROYING KEYS 184
- 8.12 PUBLIC-KEY KEY MANAGEMENT 185

9 ALGORITHM TYPES AND MODES 189

- 9.1 ELECTRONIC CODEBOOK MODE 189
- 9.2 BLOCK REPLAY 191
- 9.3 CIPHER BLOCK CHAINING MODE 193
- 9.4 STREAM CIPHERS 197
- 9.5 SELF-SYNCHRONIZING STREAM CIPHERS 198
- 9.6 CIPHER-FEEDBACK MODE 200
- 9.7 SYNCHRONOUS STREAM CIPHERS 202
- 9.8 OUTPUT-FEEDBACK MODE 203
- 9.9 COUNTER MODE 205
- 9.10 OTHER BLOCK-CIPHER MODES 206
- 9.11 CHOOSING A CIPHER MODE 208
- 9.12 INTERLEAVING 210
- 9.13 BLOCK CIPHERS VERSUS STREAM CIPHERS 210

10 USING ALGORITHMS 213

- 10.1 CHOOSING AN ALGORITHM 214
- 10.2 PUBLIC-KEY CRYPTOGRAPHY VERSUS SYMMETRIC CRYPTOGRAPHY 216
- 10.3 ENCRYPTING COMMUNICATIONS CHANNELS 216
- 10.4 ENCRYPTING DATA FOR STORAGE 220
- 10.5 HARDWARE ENCRYPTION VERSUS SOFTWARE ENCRYPTION 223

- 10.6 COMPRESSION, ENCODING, AND ENCRYPTION 226
- 10.7 DETECTING ENCRYPTION 226
- 10.8 HIDING CIPHERTEXT IN CIPHERTEXT 227
- 10.9 DESTROYING INFORMATION 228

PART III CRYPTOGRAPHIC ALGORITHMS

11 MATHEMATICAL BACKGROUND 233

- 11.1 INFORMATION THEORY 233
- 11.2 COMPLEXITY THEORY 237
- 11.3 NUMBER THEORY 242
- 11.4 FACTORING 255
- 11.5 PRIME NUMBER GENERATION 258
- 11.6 DISCRETE LOGARITHMS IN A FINITE FIELD 261

12 DATA ENCRYPTION STANDARD (DES) 265

- 12.1 BACKGROUND 265
- 12.2 DESCRIPTION OF DES 270
- 12.3 SECURITY OF DES 278
- 12.4 DIFFERENTIAL AND LINEAR CRYPTANALYSIS 285
- 12.5 THE REAL DESIGN CRITERIA 293
- 12.6 DES VARIANTS 294
- 12.7 HOW SECURE IS DES TODAY? 300

13 OTHER BLOCK CIPHERS 303

- 13.1 LUCIFER 303
- 13.2 MADRYGA 304
- 13.3 NEWDES 306
- 13.4 FEAL 308
- 13.5 REDOC 311
- 13.6 LOKI 314
- 13.7 KHUFU AND KHAFRE 316
- 13.8 RC2 318
- 13.9 IDEA 319
- 13.10 MMB 325
- 13.11 CA-1.1 327
- 13.12 SKIPJACK 328

14 STILL OTHER BLOCK CIPHERS 331

- 14.1 GOST 331
- 14.2 CAST 334
- 14.3 BLOWFISH 336
- 14.4 SAFER 339
- 14.5 3-WAY 341

- 14.6 CRAB 342
- 14.7 SXAL8/MBAL 344
- 14.8 RC5 344
- 14.9 OTHER BLOCK ALGORITHMS 346
- 14.10 THEORY OF BLOCK CIPHER DESIGN 346
- 14.11 USING ONE-WAY HASH FUNCTIONS 351
- 14.12 CHOOSING A BLOCK ALGORITHM 354

15 COMBINING BLOCK CIPHERS 357

- 15.1 DOUBLE ENCRYPTION 357
- 15.2 TRIPLE ENCRYPTION 358
- 15.3 DOUBLING THE BLOCK LENGTH 363
- 15.4 OTHER MULTIPLE ENCRYPTION SCHEMES 363
- 15.5 CDMF KEY SHORTENING 366
- 15.6 WHITENING 366
- 15.7 CASCADING MULTIPLE BLOCK ALGORITHMS 367
- 15.8 COMBINING MULTIPLE BLOCK ALGORITHMS 368

16 PSEUDO-RANDOM-SEQUENCE GENERATORS AND STREAM CIPHERS 369

- 16.1 LINEAR CONGRUENTIAL GENERATORS 369
- 16.2 LINEAR FEEDBACK SHIFT REGISTERS 372
- 16.3 DESIGN AND ANALYSIS OF STREAM CIPHERS 379
- 16.4 STREAM CIPHERS USING LFSRS 381
- 16.5 A5 389
- 16.6 HUGHES XPD/KPD 389
- 16.7 NANOTEQ 390
- 16.8 RAMBUTAN 390
- 16.9 ADDITIVE GENERATORS 390
- 16.10 GIFFORD 392
- 16.11 ALGORITHM M 393
- 16.12 PKZIP 394

17 OTHER STREAM CIPHERS AND REAL RANDOM-SEQUENCE GENERATORS 397

- 17.1 RC4 397
- 17.2 SEAL 398
- 17.3 WAKE 400
- 17.4 FEEDBACK WITH CARRY SHIFT REGISTERS 402
- 17.5 STREAM CIPHERS USING FCSRs 405
- 17.6 NONLINEAR-FEEDBACK SHIFT REGISTERS 412
- 17.7 OTHER STREAM CIPHERS 413
- 17.8 SYSTEM-THEORETIC APPROACH TO STREAM-CIPHER DESIGN 415
- 17.9 COMPLEXITY-THEMATIC APPROACH TO STREAM-CIPHER DESIGN 416
- 17.10 OTHER APPROACHES TO STREAM-CIPHER DESIGN 418

17.11	CASCADING MULTIPLE STREAM CIPHERS	419
17.12	CHOOSING A STREAM CIPHER	420
17.13	GENERATING MULTIPLE STREAMS FROM A SINGLE PSEUDO-RANDOM-SEQUENCE GENERATOR	420
17.14	REAL RANDOM-SEQUENCE GENERATORS	421
18 ONE-WAY HASH FUNCTIONS		429
18.1	BACKGROUND	429
18.2	SNEFRU	431
18.3	N-HASH	432
18.4	MD4	435
18.5	MD5	436
18.6	MD2	441
18.7	SECURE HASH ALGORITHM (SHA)	441
18.8	RIPE-MD	445
18.9	HAVAL	445
18.10	OTHER ONE-WAY HASH FUNCTIONS	446
18.11	ONE-WAY HASH FUNCTIONS USING SYMMETRIC BLOCK ALGORITHMS	446
18.12	USING PUBLIC-KEY ALGORITHMS	455
18.13	CHOOSING A ONE-WAY HASH FUNCTION	455
18.14	MESSAGE AUTHENTICATION CODES	455
19 PUBLIC-KEY ALGORITHMS		461
19.1	BACKGROUND	461
19.2	KNAPSACK ALGORITHMS	462
19.3	RSA	466
19.4	POHLIG-HELLMAN	474
19.5	RABIN	475
19.6	ELGAMAL	476
19.7	McELIECE	479
19.8	ELLIPTIC CURVE CRYPTOSYSTEMS	480
19.9	LUC	481
19.10	FINITE AUTOMATION PUBLIC-KEY CRYPTOSYSTEMS	482
20 PUBLIC-KEY DIGITAL SIGNATURE ALGORITHMS		483
20.1	DIGITAL SIGNATURE ALGORITHM (DSA)	483
20.2	DSA VARIANTS	494
20.3	GOST DIGITAL SIGNATURE ALGORITHM	495
20.4	DISCRETE LOGARITHM SIGNATURE SCHEMES	496
20.5	ONG-SCHNORR-SHAMIR	498
20.6	ESIGN	499
20.7	CELLULAR AUTOMATA	500
20.8	OTHER PUBLIC-KEY ALGORITHMS	500
21 IDENTIFICATION SCHEMES		503
21.1	FEIGE-FIAT-SHAMIR	503

21.2	GUILLOU-QUISQUATER	508
21.3	SCHNORR	510
21.4	CONVERTING IDENTIFICATION SCHEMES TO SIGNATURE SCHEMES	512
 22 KEY-EXCHANGE ALGORITHMS		
22.1	DIFFIE-HELLMAN	513
22.2	STATION-TO-STATION PROTOCOL	516
22.3	SHAMIR'S THREE-PASS PROTOCOL	516
22.4	COMSET	517
22.5	ENCRYPTED KEY EXCHANGE	518
22.6	FORTIFIED KEY NEGOTIATION	522
22.7	CONFERENCE KEY DISTRIBUTION AND SECRET BROADCASTING	523
 23 SPECIAL ALGORITHMS FOR PROTOCOLS		
23.1	MULTIPLE-KEY PUBLIC-KEY CRYPTOGRAPHY	527
23.2	SECRET-SHARING ALGORITHMS	528
23.3	SUBLIMINAL CHANNEL	531
23.4	UNDENIABLE DIGITAL SIGNATURES	536
23.5	DESIGNATED CONFIRMER SIGNATURES	539
23.6	COMPUTING WITH ENCRYPTED DATA	540
23.7	FAIR COIN FLIPS	541
23.8	ONE-WAY ACCUMULATORS	543
23.9	ALL-OR-NOTHING DISCLOSURE OF SECRETS	543
23.10	FAIR AND FAILSAFE CRYPTOSYSTEMS	546
23.11	ZERO-KNOWLEDGE PROOFS OF KNOWLEDGE	548
23.12	BLIND SIGNATURES	549
23.13	OBLIVIOUS TRANSFER	550
23.14	SECURE MULTIPARTY COMPUTATION	551
23.15	PROBABILISTIC ENCRYPTION	552
23.16	QUANTUM CRYPTOGRAPHY	554

PART IV THE REAL WORLD

24 EXAMPLE IMPLEMENTATIONS		561
24.1	IBM SECRET-KEY MANAGEMENT PROTOCOL	561
24.2	MITRENET	562
24.3	ISDN	563
24.4	STU-III	565
24.5	KERBEROS	566
24.6	KRYPTOKNIGHT	571
24.7	SESAME	572
24.8	IBM COMMON CRYPTOGRAPHIC ARCHITECTURE	573
24.9	ISO AUTHENTICATION FRAMEWORK	574
24.10	PRIVACY-ENHANCED MAIL (PEM)	577
24.11	MESSAGE SECURITY PROTOCOL (MSP)	584

- 24.12 PRETTY GOOD PRIVACY (PGP) 584
- 24.13 SMART CARDS 587
- 24.14 PUBLIC-KEY CRYPTOGRAPHY STANDARDS (PKCS) 588
- 24.15 UNIVERSAL ELECTRONIC PAYMENT SYSTEM (UEPS) 589
- 24.16 CLIPPER 591
- 24.17 CAPSTONE 593
- 24.18 AT&T MODEL 3600 TELEPHONE SECURITY DEVICE (TSD) 594

25 POLITICS 597

- 25.1 NATIONAL SECURITY AGENCY (NSA) 597
- 25.2 NATIONAL COMPUTER SECURITY CENTER (NCSC) 599
- 25.3 NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST) 600
- 25.4 RSA DATA SECURITY, INC. 603
- 25.5 PUBLIC KEY PARTNERS 604
- 25.6 INTERNATIONAL ASSOCIATION FOR CRYPTOGRAPHIC RESEARCH (IACR) 605
- 25.7 RACE INTEGRITY PRIMITIVES EVALUATION (RIPE) 605
- 25.8 CONDITIONAL ACCESS FOR EUROPE (CAFE) 606
- 25.9 ISO/IEC 9979 607
- 25.10 PROFESSIONAL, CIVIL LIBERTIES, AND INDUSTRY GROUPS 608
- 25.11 SCLCRYPT 608
- 25.12 CYPHERPUNKS 609
- 25.13 PATENTS 609
- 25.14 U.S. EXPORT RULES 610
- 25.15 FOREIGN IMPORT AND EXPORT OF CRYPTOGRAPHY 617
- 25.16 LEGAL ISSUES 618

Afterword by Matt Blaze 619

PART V SOURCE CODE

Source Code 623

References 679

Foreword

By Whitfield Diffie

The literature of cryptography has a curious history. Secrecy, of course, has always played a central role, but until the First World War, important developments appeared in print in a more or less timely fashion and the field moved forward in much the same way as other specialized disciplines. As late as 1918, one of the most influential cryptanalytic papers of the twentieth century, William F. Friedman's monograph *The Index of Coincidence and Its Applications in Cryptography*, appeared as a research report of the private Riverbank Laboratories [577]. And this, despite the fact that the work had been done as part of the war effort. In the same year Edward H. Hebern of Oakland, California filed the first patent for a rotor machine [710], the device destined to be a mainstay of military cryptography for nearly 50 years.

After the First World War, however, things began to change. U.S. Army and Navy organizations, working entirely in secret, began to make fundamental advances in cryptography. During the thirties and forties a few basic papers did appear in the open literature and several treatises on the subject were published, but the latter were farther and farther behind the state of the art. By the end of the war the transition was complete. With one notable exception, the public literature had died. That exception was Claude Shannon's paper "The Communication Theory of Secrecy Systems," which appeared in the *Bell System Technical Journal* in 1949 [1432]. It was similar to Friedman's 1918 paper, in that it grew out of wartime work of Shannon's. After the Second World War ended it was declassified, possibly by mistake.

From 1949 until 1967 the cryptographic literature was barren. In that year a different sort of contribution appeared: David Kahn's history, *The Codebreakers* [794]. It didn't contain any new technical ideas, but it did contain a remarkably complete history of what had gone before, including mention of some things that the government still considered secret. The significance of *The Codebreakers* lay not just in its remarkable scope, but also in the fact that it enjoyed good sales and made tens of thousands of people, who had never given the matter a moment's thought, aware of cryptography. A trickle of new cryptographic papers began to be written.

At about the same time, Horst Feistel, who had earlier worked on identification friend or foe devices for the Air Force, took his lifelong passion for cryptography to the IBM Watson Laboratory in Yorktown Heights, New York. There, he began development of what was to become the U.S. Data Encryption Standard; by the early 1970s several technical reports on this subject by Feistel and his colleagues had been made public by IBM [1482,1484,552].

This was the situation when I entered the field in late 1972. The cryptographic literature wasn't abundant, but what there was included some very shiny nuggets.

Cryptology presents a difficulty not found in normal academic disciplines: the need for the proper interaction of cryptography and cryptanalysis. This arises out of the fact that in the absence of real communications requirements, it is easy to propose a system that appears unbreakable. Many academic designs are so complex that the would-be cryptanalyst doesn't know where to start; exposing flaws in these designs is far harder than designing them in the first place. The result is that the competitive process, which is one strong motivation in academic research, cannot take hold.

When Martin Hellman and I proposed public-key cryptography in 1975 [496], one of the indirect aspects of our contribution was to introduce a problem that does not even appear easy to solve. Now an aspiring cryptosystem designer could produce something that would be recognized as clever—something that did more than just turn meaningful text into nonsense. The result has been a spectacular increase in the number of people working in cryptography, the number of meetings held, and the number of books and papers published.

In my acceptance speech for the Donald E. Fink award—given for the best expository paper to appear in an IEEE journal—which I received jointly with Hellman in 1980, I told the audience that in writing "Privacy and Authentication," I had an experience that I suspected was rare even among the prominent scholars who populate the IEEE awards ceremony: I had written the paper I had wanted to study, but could not find, when I first became seriously interested in cryptography. Had I been able to go to the Stanford bookstore and pick up a modern cryptography text, I would probably have learned about the field years earlier. But the only things available in the fall of 1972 were a few classic papers and some obscure technical reports.

The contemporary researcher has no such problem. The problem now is choosing where to start among the thousands of papers and dozens of books. The contemporary researcher, yes, but what about the contemporary programmer or engineer who merely wants to use cryptography? Where does that person turn? Until now, it has been necessary to spend long hours hunting out and then studying the research literature before being able to design the sort of cryptographic utilities glibly described in popular articles.

This is the gap that Bruce Schneier's *Applied Cryptography* has come to fill. Beginning with the objectives of communication security and elementary examples of programs used to achieve these objectives, Schneier gives us a panoramic view of the fruits of 20 years of public research. The title says it all; from the mundane objective of having a secure conversation the very first time you call someone to the possibilities of digital money and cryptographically secure elections, this is where you'll find it.

Not satisfied that the book was about the real world merely because it went all the way down to the code, Schneier has included an account of the world in which cryptography is developed and applied, and discusses entities ranging from the International Association for Cryptologic Research to the NSA.

When public interest in cryptography was just emerging in the late seventies and early eighties, the National Security Agency (NSA), America's official cryptographic organ, made several attempts to quash it. The first was a letter from a long-time NSA employee allegedly, avowedly, and apparently acting on his own. The letter was sent to the IEEE and warned that the publication of cryptographic material was a violation of the International Traffic in Arms Regulations (ITAR). This viewpoint turned out not even to be supported by the regulations themselves—which contained an explicit exemption for published material—but gave both the public practice of cryptography and the 1977 Information Theory Workshop lots of unexpected publicity.

A more serious attempt occurred in 1980, when the NSA funded the American Council on Education to examine the issue with a view to persuading Congress to give it legal control of publications in the field of cryptography. The results fell far short of NSA's ambitions and resulted in a program of voluntary review of cryptographic papers; researchers were requested to ask the NSA's opinion on whether disclosure of results would adversely affect the national interest before publication.

As the eighties progressed, pressure focused more on the practice than the study of cryptography. Existing laws gave the NSA the power, through the Department of State, to regulate the export of cryptographic equipment. As business became more and more international and the American fraction of the world market declined, the pressure to have a single product in both domestic and offshore markets increased. Such single products were subject to export control and thus the NSA acquired substantial influence not only over what was exported, but also over what was sold in the United States.

As this is written, a new challenge confronts the public practice of cryptography. The government has augmented the widely published and available Data Encryption Standard, with a secret algorithm implemented in tamper-resistant chips. These chips will incorporate a codified mechanism of government monitoring. The negative aspects of this "key-escrow" program range from a potentially disastrous impact on personal privacy to the high cost of having to add hardware to products that had previously encrypted in software. So far key escrow products are enjoying less than stellar sales and the scheme has attracted widespread negative comment, especially from the independent cryptographers. Some people, however, see more future in programming than politicking and have redoubled their efforts to provide the world with strong cryptography that is accessible to public scrutiny.

A sharp step back from the notion that export control law could supersede the First Amendment seemed to have been taken in 1980 when the *Federal Register* announcement of a revision to ITAR included the statement: "... provision has been added to make it clear that the regulation of the export of technical data does not purport to interfere with the First Amendment rights of individuals." But the fact that tension between the First Amendment and the export control laws has not

gone away should be evident from statements at a conference held by RSA Data Security. NSA's representative from the export control office expressed the opinion that people who published cryptographic programs were "in a grey area" with respect to the law. If that is so, it is a grey area on which the first edition of this book has shed some light. Export applications for the book itself have been granted, with acknowledgement that published material lay beyond the authority of the Munitions Control Board. Applications to export the enclosed programs on disk, however, have been denied.

The shift in the NSA's strategy, from attempting to control cryptographic research to tightening its grip on the development and deployment of cryptographic products, is presumably due to its realization that all the great cryptographic papers in the world do not protect a single bit of traffic. Sitting on the shelf, this volume may be able to do no better than the books and papers that preceded it, but sitting next to a workstation, where a programmer is writing cryptographic code, it just may.

Whitfield Diffie
Mountain View, CA

Preface

There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files. This book is about the latter.

If I take a letter, lock it in a safe, hide the safe somewhere in New York, then tell you to read the letter, that's not security. That's obscurity. On the other hand, if I take a letter and lock it in a safe, and then give you the safe along with the design specifications of the safe and a hundred identical safes with their combinations so that you and the world's best safecrackers can study the locking mechanism—and you still can't open the safe and read the letter—that's security.

For many years, this sort of cryptography was the exclusive domain of the military. The United States' National Security Agency (NSA), and its counterparts in the former Soviet Union, England, France, Israel, and elsewhere, have spent billions of dollars in the very serious game of securing their own communications while trying to break everyone else's. Private individuals, with far less expertise and budget, have been powerless to protect their own privacy against these governments.

During the last 20 years, public academic research in cryptography has exploded. While classical cryptography has been long used by ordinary citizens, computer cryptography was the exclusive domain of the world's militaries since World War II. Today, state-of-the-art computer cryptography is practiced outside the secured walls of the military agencies. The layperson can now employ security practices that can protect against the most powerful of adversaries—security that may protect against military agencies for years to come.

Do average people really need this kind of security? Yes. They may be planning a political campaign, discussing taxes, or having an illicit affair. They may be designing a new product, discussing a marketing strategy, or planning a hostile business takeover. Or they may be living in a country that does not respect the rights of privacy of its citizens. They may be doing something that they feel shouldn't be illegal,

but is. For whatever reason, the data and communications are personal, private, and no one else's business.

This book is being published in a tumultuous time. In 1994, the Clinton administration approved the Escrowed Encryption Standard (including the Clipper chip and Fortezza card) and signed the Digital Telephony bill into law. Both of these initiatives try to ensure the government's ability to conduct electronic surveillance.

Some dangerously Orwellian assumptions are at work here: that the government has the right to listen to private communications, and that there is something wrong with a private citizen trying to keep a secret from the government. Law enforcement has always been able to conduct court-authorized surveillance if possible, but this is the first time that the people have been forced to take active measures to *make themselves available* for surveillance. These initiatives are not simply government proposals in some obscure area; they are preemptive and unilateral attempts to usurp powers that previously belonged to the people.

Clipper and Digital Telephony do not protect privacy; they force individuals to unconditionally trust that the government will respect their privacy. The same law enforcement authorities who illegally tapped Martin Luther King Jr.'s phones can easily tap a phone protected with Clipper. In the recent past, local police authorities have either been charged criminally or sued civilly in numerous jurisdictions—Maryland, Connecticut, Vermont, Georgia, Missouri, and Nevada—for conducting illegal wiretaps. It's a poor idea to deploy a technology that could some day facilitate a police state.

The lesson here is that it is insufficient to protect ourselves with laws; we need to protect ourselves with mathematics. Encryption is too important to be left solely to governments.

This book gives you the tools you need to protect your own privacy; cryptography products may be declared illegal, but the information will never be.

HOW TO READ THIS BOOK

I wrote *Applied Cryptography* to be both a lively introduction to the field of cryptography and a comprehensive reference. I have tried to keep the text readable without sacrificing accuracy. This book is not intended to be a mathematical text. Although I have not deliberately given any false information, I do play fast and loose with theory. For those interested in formalism, there are copious references to the academic literature.

Chapter 1 introduces cryptography, defines many terms, and briefly discusses pre-computer cryptography.

Chapters 2 through 6 (Part I) describe cryptographic protocols: what people can do with cryptography. The protocols range from the simple (sending encrypted messages from one person to another) to the complex (flipping a coin over the telephone) to the esoteric (secure and anonymous digital money exchange). Some of these protocols are obvious; others are almost amazing. Cryptography can solve a lot of problems that most people never realized it could.

Chapters 7 through 10 (Part II) discuss cryptographic techniques. All four chapters in this section are important for even the most basic uses of cryptography. Chapters 7 and 8 are about keys: how long a key should be in order to be secure, how to generate keys, how to store keys, how to dispose of keys, and so on. Key management is the hardest part of cryptography and often the Achilles' heel of an otherwise secure system. Chapter 9 discusses different ways of using cryptographic algorithms, and Chapter 10 gives the odds and ends of algorithms: how to choose, implement, and use algorithms.

Chapters 11 through 23 (Part III) list algorithms. Chapter 11 provides the mathematical background. This chapter is only required if you are interested in public-key algorithms. If you just want to implement DES (or something similar), you can skip ahead. Chapter 12 discusses DES: the algorithm, its history, its security, and some variants. Chapters 13, 14, and 15 discuss other block algorithms; if you want something more secure than DES, skip to the section on IDEA and triple-DES. If you want to read about a bunch of algorithms, some of which may be more secure than DES, read the whole chapter. Chapters 16 and 17 discuss stream algorithms. Chapter 18 focuses on one-way hash functions; MD5 and SHA are the most common, although I discuss many more. Chapter 19 discusses public-key encryption algorithms, Chapter 20 discusses public-key digital signature algorithms, Chapter 21 discusses public-key identification algorithms, and Chapter 22 discusses public-key key exchange algorithms. The important algorithms are RSA, DSA, Fiat-Shamir, and Diffie-Hellman, respectively. Chapter 23 has more esoteric public-key algorithms and protocols; the math in this chapter is quite complicated, so wear your seat belt.

Chapters 24 and 25 (Part IV) turn to the real world of cryptography. Chapter 24 discusses some of the current implementations of these algorithms and protocols, while Chapter 25 touches on some of the political issues surrounding cryptography. These chapters are by no means intended to be comprehensive.

Also included are source code listings for 10 algorithms discussed in Part III. I was unable to include all the code I wanted to due to space limitations, and cryptographic source code cannot otherwise be exported. (Amazingly enough, the State Department allowed export of the first edition of this book with source code, but denied export for a computer disk with the exact same source code on it. Go figure.) An associated source code disk set includes much more source code than I could fit in this book; it is probably the largest collection of cryptographic source code outside a military institution. I can only send source code disks to U.S. and Canadian citizens living in the U.S. and Canada, but hopefully that will change someday. If you are interested in implementing or playing with the cryptographic algorithms in this book, get the disk. See the last page of the book for details.

One criticism of this book is that its encyclopedic nature takes away from its readability. This is true, but I wanted to provide a single reference for those who might come across an algorithm in the academic literature or in a product. For those who are more interested in a tutorial, I apologize. A lot is being done in the field; this is the first time so much of it has been gathered between two covers. Even so, space considerations forced me to leave many things out. I covered topics that I felt were important, practical, or interesting. If I couldn't cover a topic in depth, I gave references to articles and papers that did.

I have done my best to hunt down and eradicate all errors in this book, but many have assured me that it is an impossible task. Certainly, the second edition has far fewer errors than the first. An errata listing is available from me and will be periodically posted to the Usenet newsgroup sci.crypt. If any reader finds an error, please let me know. I'll send the first person to find each error in the book a free copy of the source code disk.

Acknowledgments

The list of people who had a hand in this book may seem unending, but all are worthy of mention. I would like to thank Don Alvarez, Ross Anderson, Dave Balenson, Karl Barrus, Steve Bellovin, Dan Bernstein, Eli Biham, Joan Boyar, Karen Cooper, Whit Diffie, Joan Feigenbaum, Phil Karn, Neal Koblitz, Xuejia Lai, Tom Leranthe, Mike Markowitz, Ralph Merkle, Bill Patton, Peter Pearson, Charles Pfleeger, Ken Pizzini, Bart Preneel, Mark Riordan, Joachim Schurman, and Marc Schwartz for reading and editing all or parts of the first edition; Marc Vauclair for translating the first edition into French; Abe Abraham, Ross Anderson, Dave Banisar, Steve Bellovin, Eli Biham, Matt Bishop, Matt Blaze, Gary Carter, Jan Comenisch, Claude Crépeau, Joan Daemen, Jorge Davila, Ed Dawson, Whit Diffie, Carl Ellison, Joan Feigenbaum, Niels Ferguson, Matt Franklin, Rosario Gennaro, Dieter Gollmann, Mark Goresky, Richard Graveman, Stuart Haber, Jingman He, Bob Hogue, Kenneth Iversen, Markus Jakobsson, Burt Kaliski, Phil Karn, John Kelsey, John Kennedy, Lars Knudsen, Paul Kocher, John Ladwig, Xuejia Lai, Arjen Lenstra, Paul Leyland, Mike Markowitz, Jim Massey, Bruce McNair, William Hugh Murray, Roger Needham, Clif Neuman, Kaisa Nyberg, Luke O'Connor, Peter Pearson, René Peralta, Bart Preneel, Yisrael Radai, Matt Robshaw, Michael Roe, Phil Rogaway, Avi Rubin, Paul Rubin, Selwyn Russell, Kazue Sako, Mahmoud Salmasizadeh, Markus Stadler, Dmitry Titov, Jimmy Upton, Marc Vauclair, Serge Vaude- nay, Gideon Yuval, Glen Zorn, and several anonymous government employees for reading and editing all or parts of the second edition; Lawrie Brown, Leisa Condie, Joan Daemen, Peter Gutmann, Alan Insley, Chris Johnston, John Kelsey, Xuejia Lai, Bill Leininger, Mike Markowitz, Richard Outerbridge, Peter Pearson, Ken Pizzini, Colin Plumb, RSA Data Security, Inc., Michael Roe, Michael Wood, and Phil Zimmermann for providing source code; Paul MacNerland for creating the figures for the first edition; Karen Cooper for copyediting the second edition; Beth Friedman for proofreading the second edition; Carol Kennedy for indexing the second edition; the readers of sci.crypt and the Cypherpunks mailing list for commenting on ideas, answering questions, and finding errors in the first edition; Randy Seuss for providing Internet access; Jeff Duntemann and Jon Erickson for helping me get started; assorted random Insleys for the impetus, encouragement, support, conversations, friendship, and dinners; and AT&T Bell Labs for firing me and making this all possible. All these people helped to create a far better book than I could have created alone.

Bruce Schneier
Oak Park, Ill.
schneier@counterpane.com

About the Author

BRUCE SCHNEIER is president of Counterpane Systems, an Oak Park, Illinois consulting firm specializing in cryptography and computer security. Bruce is also the author of *E-Mail Security* (John Wiley & Sons, 1995) and *Protect Your Macintosh* (Peachpit Press, 1994); and has written dozens of articles on cryptography for major magazines. He is a contributing editor to *Dr. Dobb's Journal*, where he edits the "Algorithms Alley" column, and a contributing editor to *Computer and Communications Security Reviews*. Bruce serves on the board of directors of the International Association for Cryptologic Research, is a member of the Advisory Board for the Electronic Privacy Information Center, and is on the program committee for the New Security Paradigms Workshop. In addition, he finds time to give frequent lectures on cryptography, computer security, and privacy.

CHAPTER 1

Foundations

1.1 TERMINOLOGY

Sender and Receiver

Suppose a sender wants to send a message to a receiver. Moreover, this sender wants to send the message securely: She wants to make sure an eavesdropper cannot read the message.

Messages and Encryption

A message is **plaintext** (sometimes called cleartext). The process of disguising a message in such a way as to hide its substance is **encryption**. An encrypted message is **ciphertext**. The process of turning ciphertext back into plaintext is **decryption**. This is all shown in Figure 1.1.

(If you want to follow the ISO 7498-2 standard, use the terms “encipher” and “decipher.” It seems that some cultures find the terms “encrypt” and “decrypt” offensive, as they refer to dead bodies.)

The art and science of keeping messages secure is **cryptography**, and it is practiced by **cryptographers**. **Cryptanalysts** are practitioners of **cryptanalysis**, the art and science of breaking ciphertext; that is, seeing through the disguise. The branch of mathematics encompassing both cryptography and cryptanalysis is **cryptology** and its practitioners are **cryptologists**. Modern cryptologists are generally trained in theoretical mathematics—they have to be.

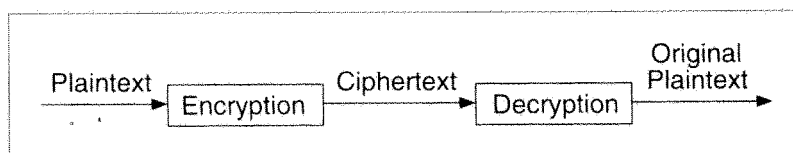


Figure 1.1 Encryption and Decryption.

Plaintext is denoted by M , for message, or P , for plaintext. It can be a stream of bits, a text file, a bitmap, a stream of digitized voice, a digital video image . . . whatever. As far as a computer is concerned, M is simply binary data. (After this chapter, this book concerns itself with binary data and computer cryptography.) The plaintext can be intended for either transmission or storage. In any case, M is the message to be encrypted.

Ciphertext is denoted by C . It is also binary data: sometimes the same size as M , sometimes larger. (By combining encryption with compression, C may be smaller than M . However, encryption does not accomplish this.) The encryption function E , operates on M to produce C . Or, in mathematical notation:

$$E(M) = C$$

In the reverse process, the decryption function D operates on C to produce M :

$$D(C) = M$$

Since the whole point of encrypting and then decrypting a message is to recover the original plaintext, the following identity must hold true:

$$D(E(M)) = M$$

Authentication, Integrity, and Nonrepudiation

In addition to providing confidentiality, cryptography is often asked to do other jobs:

- **Authentication.** It should be possible for the receiver of a message to ascertain its origin; an intruder should not be able to masquerade as someone else.
- **Integrity.** It should be possible for the receiver of a message to verify that it has not been modified in transit; an intruder should not be able to substitute a false message for a legitimate one.
- **Nonrepudiation.** A sender should not be able to falsely deny later that he sent a message.

These are vital requirements for social interaction on computers, and are analogous to face-to-face interactions. That someone is who he says he is . . . that someone's credentials—whether a driver's license, a medical degree, or a passport—are valid . . . that a document purporting to come from a person actually came from that person. . . . These are the things that authentication, integrity, and nonrepudiation provide.

Algorithms and Keys

A **cryptographic algorithm**, also called a **cipher**, is the mathematical function used for encryption and decryption. (Generally, there are two related functions: one for encryption and the other for decryption.)

If the security of an algorithm is based on keeping the way that algorithm works a secret, it is a **restricted** algorithm. Restricted algorithms have historical interest, but are woefully inadequate by today's standards. A large or changing group of users cannot use them, because every time a user leaves the group everyone else must switch to a different algorithm. If someone accidentally reveals the secret, everyone must change their algorithm.

Even more damning, restricted algorithms allow no quality control or standardization. Every group of users must have their own unique algorithm. Such a group can't use off-the-shelf hardware or software products; an eavesdropper can buy the same product and learn the algorithm. They have to write their own algorithms and implementations. If no one in the group is a good cryptographer, then they won't know if they have a secure algorithm.

Despite these major drawbacks, restricted algorithms are enormously popular for low-security applications. Users either don't realize or don't care about the security problems inherent in their system.

Modern cryptography solves this problem with a **key**, denoted by K . This key might be any one of a large number of values. The range of possible values of the key is called the **keyspace**. Both the encryption and decryption operations use this key (i.e., they are dependent on the key and this fact is denoted by the K subscript), so the functions now become:

$$E_K(M) = C$$

$$D_K(C) = M$$

Those functions have the property that (see Figure 1.2):

$$D_K(E_K(M)) = M$$

Some algorithms use a different encryption key and decryption key (see Figure 1.3). That is, the encryption key, K_1 , is different from the corresponding decryption key, K_2 . In this case:

$$E_{K_1}(M) = C$$

$$D_{K_2}(C) = M$$

$$D_{K_2}(E_{K_1}(M)) = M$$

All of the security in these algorithms is based in the key (or keys); none is based in the details of the algorithm. This means that the algorithm can be published and analyzed. Products using the algorithm can be mass-produced. It doesn't matter if an

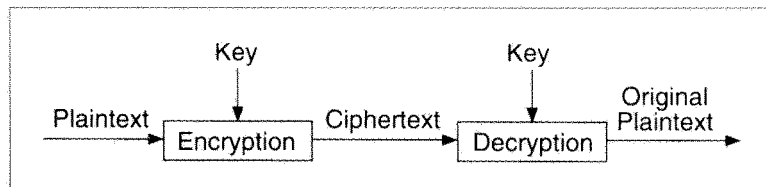


Figure 1.2 Encryption and decryption with a key.

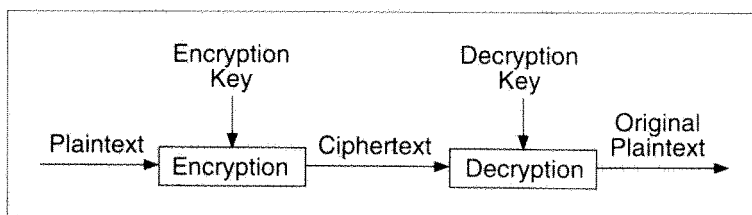


Figure 1.3 Encryption and decryption with two different keys.

eavesdropper knows your algorithm; if she doesn't know your particular key, she can't read your messages.

A **cryptosystem** is an algorithm, plus all possible plaintexts, ciphertexts, and keys.

Symmetric Algorithms

There are two general types of key-based algorithms: symmetric and public-key. **Symmetric algorithms**, sometimes called conventional algorithms, are algorithms where the encryption key can be calculated from the decryption key and vice versa. In most symmetric algorithms, the encryption key and the decryption key are the same. These algorithms, also called secret-key algorithms, single-key algorithms, or one-key algorithms, require that the sender and receiver agree on a key before they can communicate securely. The security of a symmetric algorithm rests in the key; divulging the key means that anyone could encrypt and decrypt messages. As long as the communication needs to remain secret, the key must remain secret.

Encryption and decryption with a symmetric algorithm are denoted by:

$$E_K(M) = C$$

$$D_K(C) = M$$

Symmetric algorithms can be divided into two categories. Some operate on the plaintext a single bit (or sometimes byte) at a time; these are called **stream algorithms** or **stream ciphers**. Others operate on the plaintext in groups of bits. The groups of bits are called **blocks**, and the algorithms are called **block algorithms** or **block ciphers**. For modern computer algorithms, a typical block size is 64 bits—large enough to preclude analysis and small enough to be workable. (Before computers, algorithms generally operated on plaintext one character at a time. You can think of this as a stream algorithm operating on a stream of characters.)

Public-Key Algorithms

Public-key algorithms (also called asymmetric algorithms) are designed so that the key used for encryption is different from the key used for decryption. Furthermore, the decryption key cannot (at least in any reasonable amount of time) be calculated from the encryption key. The algorithms are called “public-key” because the encryption key can be made public: A complete stranger can use the encryption key to encrypt a message, but only a specific person with the corresponding decryp-

tion key can decrypt the message. In these systems, the encryption key is often called the **public key**, and the decryption key is often called the **private key**. The private key is sometimes also called the secret key, but to avoid confusion with symmetric algorithms, that tag won't be used here.

Encryption using public key K is denoted by:

$$E_K(M) = C$$

Even though the public key and private key are different, decryption with the corresponding private key is denoted by:

$$D_K(C) = M$$

Sometimes, messages will be encrypted with the private key and decrypted with the public key; this is used in digital signatures (see Section 2.6). Despite the possible confusion, these operations are denoted by, respectively:

$$E_K(M) = C$$

$$D_K(C) = M$$

Cryptanalysis

The whole point of cryptography is to keep the plaintext (or the key, or both) secret from eavesdroppers (also called adversaries, attackers, interceptors, interlopers, intruders, opponents, or simply the enemy). Eavesdroppers are assumed to have complete access to the communications between the sender and receiver.

Cryptanalysis is the science of recovering the plaintext of a message without access to the key. Successful cryptanalysis may recover the plaintext or the key. It also may find weaknesses in a cryptosystem that eventually lead to the previous results. (The loss of a key through noncryptanalytic means is called a **compromise**.)

An attempted cryptanalysis is called an **attack**. A fundamental assumption in cryptanalysis, first enunciated by the Dutchman A. Kerckhoffs in the nineteenth century, is that the secrecy must reside entirely in the key [794]. Kerckhoffs assumes that the cryptanalyst has complete details of the cryptographic algorithm and implementation. (Of course, one would assume that the CIA does not make a habit of telling Mossad about its cryptographic algorithms, but Mossad probably finds out anyway.) While real-world cryptanalysts don't always have such detailed information, it's a good assumption to make. If others can't break an algorithm, even with knowledge of how it works, then they certainly won't be able to break it without that knowledge.

There are four general types of cryptanalytic attacks. Of course, each of them assumes that the cryptanalyst has complete knowledge of the encryption algorithm used:

1. **Ciphertext-only attack.** The cryptanalyst has the ciphertext of several messages, all of which have been encrypted using the same encryption algorithm. The cryptanalyst's job is to recover the plaintext of as many messages as possible, or better yet to deduce the key (or keys) used to

encrypt the messages, in order to decrypt other messages encrypted with the same keys.

Given: $C_1 = E_k(P_1), C_2 = E_k(P_2), \dots, C_i = E_k(P_i)$

Deduce: Either $P_1, P_2, \dots, P_i$; k ; or an algorithm
to infer P_{i+1} from $C_{i+1} = E_k(P_{i+1})$

2. **Known-plaintext attack.** The cryptanalyst has access not only to the ciphertext of several messages, but also to the plaintext of those messages. His job is to deduce the key (or keys) used to encrypt the messages or an algorithm to decrypt any new messages encrypted with the same key (or keys).

Given: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2), \dots, P_i, C_i = E_k(P_i)$

Deduce: Either k , or an algorithm
to infer P_{i+1} from $C_{i+1} = E_k(P_{i+1})$

3. **Chosen-plaintext attack.** The cryptanalyst not only has access to the ciphertext and associated plaintext for several messages, but he also chooses the plaintext that gets encrypted. This is more powerful than a known-plaintext attack, because the cryptanalyst can choose specific plaintext blocks to encrypt, ones that might yield more information about the key. His job is to deduce the key (or keys) used to encrypt the messages or an algorithm to decrypt any new messages encrypted with the same key (or keys).

Given: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2), \dots, P_i, C_i = E_k(P_i)$,

where the cryptanalyst gets to choose $P_1, P_2, \dots, P_i$

Deduce: Either k , or an algorithm to infer P_{i+1} from $C_{i+1} = E_k(P_{i+1})$

4. **Adaptive-chosen-plaintext attack.** This is a special case of a chosen-plaintext attack. Not only can the cryptanalyst choose the plaintext that is encrypted, but he can also modify his choice based on the results of previous encryption. In a chosen-plaintext attack, a cryptanalyst might just be able to choose one large block of plaintext to be encrypted; in an adaptive-chosen-plaintext attack he can choose a smaller block of plaintext and then choose another based on the results of the first, and so forth.

There are at least three other types of cryptanalytic attack.

5. **Chosen-ciphertext attack.** The cryptanalyst can choose different ciphertexts to be decrypted and has access to the decrypted plaintext. For example, the cryptanalyst has access to a tamperproof box that does automatic decryption. His job is to deduce the key.

Given: $C_1, P_1 = D_k(C_1), C_2, P_2 = D_k(C_2), \dots, C_i, P_i = D_k(C_i)$

Deduce: k

This attack is primarily applicable to public-key algorithms and will be discussed in Section 19.3. A chosen-ciphertext attack is sometimes effective against a symmetric algorithm as well. (Sometimes a chosen-plaintext attack and a chosen-ciphertext attack are together known as a **chosen-text attack**.)

6. **Chosen-key attack.** This attack doesn't mean that the cryptanalyst can choose the key; it means that he has some knowledge about the relationship between different keys. It's strange and obscure, not very practical, and discussed in Section 12.4.
7. **Rubber-hose cryptanalysis.** The cryptanalyst threatens, blackmails, or tortures someone until they give him the key. Bribery is sometimes referred to as a **purchase-key attack**. These are all very powerful attacks and often the best way to break an algorithm.

Known-plaintext attacks and chosen-plaintext attacks are more common than you might think. It is not unheard-of for a cryptanalyst to get a plaintext message that has been encrypted or to bribe someone to encrypt a chosen message. You may not even have to bribe someone; if you give a message to an ambassador, you will probably find that it gets encrypted and sent back to his country for consideration. Many messages have standard beginnings and endings that might be known to the cryptanalyst. Encrypted source code is especially vulnerable because of the regular appearance of keywords: `#define`, `struct`, `else`, `return`. Encrypted executable code has the same kinds of problems: functions, loop structures, and so on. Known-plaintext attacks (and even chosen-plaintext attacks) were successfully used against both the Germans and the Japanese during World War II. David Kahn's books [794,795,796] have historical examples of these kinds of attacks.

And don't forget Kerckhoffs's assumption: If the strength of your new cryptosystem relies on the fact that the attacker does not know the algorithm's inner workings, you're sunk. If you believe that keeping the algorithm's insides secret improves the security of your cryptosystem more than letting the academic community analyze it, you're wrong. And if you think that someone won't disassemble your source code and reverse-engineer your algorithm, you're naïve. (In 1994 this happened with the RC4 algorithm—see Section 17.1.) The best algorithms we have are the ones that have been made public, have been attacked by the world's best cryptographers for years, and are still unbreakable. (The National Security Agency keeps their algorithms secret from outsiders, but they have the best cryptographers in the world working within their walls—you don't. Additionally, they discuss their algorithms with one another, relying on peer review to uncover any weaknesses in their work.)

Cryptanalysts don't always have access to the algorithms, as when the United States broke the Japanese diplomatic code PURPLE during World War II [794]—but they often do. If the algorithm is being used in a commercial security program, it is simply a matter of time and money to disassemble the program and recover the algorithm. If the algorithm is being used in a military communications system, it is sim-

ply a matter of time and money to buy (or steal) the equipment and reverse-engineer the algorithm.

Those who claim to have an unbreakable cipher simply because they can't break it are either geniuses or fools. Unfortunately, there are more of the latter in the world. Beware of people who extol the virtues of their algorithms, but refuse to make them public; trusting their algorithms is like trusting snake oil.

Good cryptographers rely on peer review to separate the good algorithms from the bad.

Security of Algorithms

Different algorithms offer different degrees of security; it depends on how hard they are to break. If the cost required to break an algorithm is greater than the value of the encrypted data, then you're probably safe. If the time required to break an algorithm is longer than the time the encrypted data must remain secret, then you're probably safe. If the amount of data encrypted with a single key is less than the amount of data necessary to break the algorithm, then you're probably safe.

I say "probably" because there is always a chance of new breakthroughs in cryptanalysis. On the other hand, the value of most data decreases over time. It is important that the value of the data always remain less than the cost to break the security protecting it.

Lars Knudsen classified these different categories of breaking an algorithm. In decreasing order of severity [858]:

1. **Total break.** A cryptanalyst finds the key, K , such that $D_K(C) = P$.
2. **Global deduction.** A cryptanalyst finds an alternate algorithm, A , equivalent to $D_K(C)$, without knowing K .
3. **Instance (or local) deduction.** A cryptanalyst finds the plaintext of an intercepted ciphertext.
4. **Information deduction.** A cryptanalyst gains some information about the key or plaintext. This information could be a few bits of the key, some information about the form of the plaintext, and so forth.

An algorithm is **unconditionally secure** if, no matter how much ciphertext a cryptanalyst has, there is not enough information to recover the plaintext. In point of fact, only a one-time pad (see Section 1.5) is unbreakable given infinite resources. All other cryptosystems are breakable in a ciphertext-only attack, simply by trying every possible key one by one and checking whether the resulting plaintext is meaningful. This is called a **brute-force** attack (see Section 7.1).

Cryptography is more concerned with cryptosystems that are computationally infeasible to break. An algorithm is considered **computationally secure** (sometimes called strong) if it cannot be broken with available resources, either current or future. Exactly what constitutes "available resources" is open to interpretation.

You can measure the complexity (see Section 11.1) of an attack in different ways:

1. **Data complexity.** The amount of data needed as input to the attack.
2. **Processing complexity.** The time needed to perform the attack. This is often called the **work factor**.
3. **Storage requirements.** The amount of memory needed to do the attack.

As a rule of thumb, the complexity of an attack is taken to be the maximum of these three factors. Some attacks involve trading off the three complexities: A faster attack might be possible at the expense of a greater storage requirement.

Complexities are expressed as orders of magnitude. If an algorithm has a processing complexity of 2^{128} , then 2^{128} operations are required to break the algorithm. (These operations may be complex and time-consuming.) Still, if you assume that you have enough computing speed to perform a million operations every second and you set a million parallel processors against the task, it will still take over 10^{19} years to recover the key. That's a billion times the age of the universe.

While the complexity of an attack is constant (until some cryptanalyst finds a better attack, of course), computing power is anything but. There have been phenomenal advances in computing power during the last half-century and there is no reason to think this trend won't continue. Many cryptanalytic attacks are perfect for parallel machines: The task can be broken down into billions of tiny pieces and none of the processors need to interact with each other. Pronouncing an algorithm secure simply because it is infeasible to break, given current technology, is dicey at best. Good cryptosystems are designed to be infeasible to break with the computing power that is expected to evolve many years in the future.

Historical Terms

Historically, a **code** refers to a cryptosystem that deals with linguistic units: words, phrases, sentences, and so forth. For example, the word "OCELOT" might be the ciphertext for the entire phrase "TURN LEFT 90 DEGREES," the word "LOLIPOP" might be the ciphertext for "TURN RIGHT 90 DEGREES," and the words "BENT EAR" might be the ciphertext for "HOWITZER." Codes of this type are not discussed in this book; see [794,795]. Codes are only useful for specialized circumstances. Ciphers are useful for any circumstance. If your code has no entry for "ANTEATERS," then you can't say it. You can say anything with a cipher.

1.2 STEGANOGRAPHY

Steganography serves to hide secret messages in other messages, such that the secret's very existence is concealed. Generally the sender writes an innocuous message and then conceals a secret message on the same piece of paper. Historical tricks include invisible inks, tiny pin punctures on selected characters, minute differences between handwritten characters, pencil marks on typewritten characters, grilles which cover most of the message except for a few characters, and so on.

More recently, people are hiding secret messages in graphic images. Replace the least significant bit of the image with a message. The graphical image won't change appreciably—most graphics standards specify more gradations of color than the human eye can notice—and the message can be stripped out at the receiving end. You can store a 64-kilobyte message in a 1024×1024 grey-scale picture this way. Several public-domain programs do this sort of thing.

Peter Wayner's **mimic functions** obfuscate messages. These functions modify a message so that its statistical profile resembles that of something else: the classifieds section of *The New York Times*, a play by Shakespeare, or a newsgroup on the Internet [1584,1585]. This type of steganography won't fool a person, but it might fool some big computers scanning the Internet for interesting messages.

1.3 SUBSTITUTION CIPHERS AND TRANSPOSITION CIPHERS

Before computers, cryptography consisted of character-based algorithms. Different cryptographic algorithms either substituted characters for one another or transposed characters with one another. The better algorithms did both, many times each.

Things are more complex these days, but the philosophy remains the same. The primary change is that algorithms work on bits instead of characters. This is actually just a change in the alphabet size: from 26 elements to two elements. Most good cryptographic algorithms still combine elements of substitution and transposition.

Substitution Ciphers

A **substitution cipher** is one in which each character in the plaintext is substituted for another character in the ciphertext. The receiver inverts the substitution on the ciphertext to recover the plaintext.

In classical cryptography, there are four types of substitution ciphers:

- A **simple substitution cipher**, or **monoalphabetic cipher**, is one in which each character of the plaintext is replaced with a corresponding character of ciphertext. The cryptograms in newspapers are simple substitution ciphers.
- A **homophonic substitution cipher** is like a simple substitution cryptosystem, except a single character of plaintext can map to one of several characters of ciphertext. For example, "A" could correspond to either 5, 13, 25, or 56, "B" could correspond to either 7, 19, 31, or 42, and so on.
- A **polygram substitution cipher** is one in which blocks of characters are encrypted in groups. For example, "ABA" could correspond to "RTQ," "ABB" could correspond to "SLL," and so on.
- A **polyalphabetic substitution cipher** is made up of multiple simple substitution ciphers. For example, there might be five different simple substitution ciphers used; the particular one used changes with the position of each character of the plaintext.

The famous **Caesar Cipher**, in which each plaintext character is replaced by the character three to the right modulo 26 ("A" is replaced by "D," "B" is replaced by "E," "W" is replaced by "Z," . . . , "X" is replaced by "A," "Y" is replaced by "B," and "Z" is replaced by "C") is a simple substitution cipher. It's actually even simpler, because the ciphertext alphabet is a rotation of the plaintext alphabet and not an arbitrary permutation.

ROT13 is a simple encryption program commonly found on UNIX systems; it is also a simple substitution cipher. In this cipher, "A" is replaced by "N," "B" is replaced by "O," and so on. Every letter is rotated 13 places.

Encrypting a file twice with ROT13 restores the original file.

$$P = \text{ROT13}(\text{ROT13}(P))$$

ROT13 is not intended for security; it is often used in Usenet posts to hide potentially offensive text, to avoid giving away the solution to a puzzle, and so forth.

Simple substitution ciphers can be easily broken because the cipher does not hide the underlying frequencies of the different letters of the plaintext. All it takes is about 25 English characters before a good cryptanalyst can reconstruct the plaintext [1434]. An algorithm for solving these sorts of ciphers can be found in [578,587,1600,78,1475,1236,880]. A good computer algorithm is [703].

Homophonic substitution ciphers were used as early as 1401 by the Duchy of Mantua [794]. They are much more complicated to break than simple substitution ciphers, but still do not obscure all of the statistical properties of the plaintext language. With a known-plaintext attack, the ciphers are trivial to break. A ciphertext-only attack is harder, but only takes a few seconds on a computer. Details are in [1261].

Polygram substitution ciphers are ciphers in which groups of letters are encrypted together. The Playfair cipher, invented in 1854, was used by the British during World War I [794]. It encrypts pairs of letters together. Its cryptanalysis is discussed in [587,1475,880]. The Hill cipher is another example of a polygram substitution cipher [732]. Sometimes you see Huffman coding used as a cipher; this is an insecure polygram substitution cipher.

Polyalphabetic substitution ciphers were invented by Leon Battista in 1568 [794]. They were used by the Union army during the American Civil War. Despite the fact that they can be broken easily [819,577,587,794] (especially with the help of computers), many commercial computer security products use ciphers of this form [1387,1390,1502]. (Details on how to break this encryption scheme, as used in WordPerfect, can be found in [135,139].) The Vigenère cipher, first published in 1586, and the Beaufort cipher are also examples of polyalphabetic substitution ciphers.

Polyalphabetic substitution ciphers have multiple one-letter keys, each of which is used to encrypt one letter of the plaintext. The first key encrypts the first letter of the plaintext, the second key encrypts the second letter of the plaintext, and so on. After all the keys are used, the keys are recycled. If there were 20 one-letter keys, then every twentieth letter would be encrypted with the same key. This is called the **period** of the cipher. In classical cryptography, ciphers with longer periods were significantly harder to break than ciphers with short periods. There are computer techniques that can easily break substitution ciphers with very long periods.

A **running-key cipher**—sometimes called a book cipher—in which one text is used to encrypt another text, is another example of this sort of cipher. Even though this cipher has a period the length of the text, it can also be broken easily [576,794].

Transposition Ciphers

In a **transposition cipher** the plaintext remains the same, but the order of characters is shuffled around. In a **simple columnar transposition cipher**, the plaintext is written horizontally onto a piece of graph paper of fixed width and the ciphertext is read off vertically (see Figure 1.4). Decryption is a matter of writing the ciphertext vertically onto a piece of graph paper of identical width and then reading the plaintext off horizontally.

Cryptanalysis of these ciphers is discussed in [587,1475]. Since the letters of the ciphertext are the same as those of the plaintext, a frequency analysis on the ciphertext would reveal that each letter has approximately the same likelihood as in English. This gives a very good clue to a cryptanalyst, who can then use a variety of techniques to determine the right ordering of the letters to obtain the plaintext. Putting the ciphertext through a second transposition cipher greatly enhances security. There are even more complicated transposition ciphers, but computers can break almost all of them.

The German ADFGVX cipher, used during World War I, is a transposition cipher combined with a simple substitution. It was a very complex algorithm for its day but was broken by Georges Painvin, a French cryptanalyst [794].

Although many modern algorithms use transposition, it is troublesome because it requires a lot of memory and sometimes requires messages to be only certain lengths. Substitution is far more common.

Rotor Machines

In the 1920s, various mechanical encryption devices were invented to automate the process of encryption. Most were based on the concept of a **rotor**, a mechanical wheel wired to perform a general substitution.

A **rotor machine** has a keyboard and a series of rotors, and implements a version of the Vigenère cipher. Each rotor is an arbitrary permutation of the alphabet, has 26 positions, and performs a simple substitution. For example, a rotor might be wired

Plaintext: COMPUTER GRAPHICS MAY BE SLOW BUT AT LEAST IT'S EXPENSIVE.

COMPUTERGR
APHICSMAYB
ESLOWBUTAT
LEASTITSEX
PENSIVE

Ciphertext: CAELP OPSEE MHLAN PIOSS UCWTITSBIV EMUTE RATSG YAERB TX

Figure 1.4 Columnar transposition cipher.

to substitute "F" for "A," "U" for "B," "L" for "C," and so on. And the output pins of one rotor are connected to the input pins of the next.

For example, in a 4-rotor machine the first rotor might substitute "F" for "A," the second might substitute "Y" for "F," the third might substitute "E" for "Y," and the fourth might substitute "C" for "E"; "C" would be the output ciphertext. Then some of the rotors shift, so next time the substitutions will be different.

It is the combination of several rotors and the gears moving them that makes the machine secure. Because the rotors all move at different rates, the period for an n -rotor machine is 26^n . Some rotor machines can also have a different number of positions on each rotor, further frustrating cryptanalysis.

The best-known rotor device is the Enigma. The Enigma was used by the Germans during World War II. The idea was invented by Arthur Scherbius and Arvid Gerhard Damm in Europe. It was patented in the United States by Arthur Scherbius [1383]. The Germans beefed up the basic design considerably for wartime use.

The German Enigma had three rotors, chosen from a set of five, a plugboard that slightly permuted the plaintext, and a reflecting rotor that caused each rotor to operate on each plaintext letter twice. As complicated as the Enigma was, it was broken during World War II. First, a team of Polish cryptographers broke the German Enigma and explained their attack to the British. The Germans modified their Enigma as the war progressed, and the British continued to cryptanalyze the new versions. For explanations of how rotor ciphers work and how they were broken, see [794,86,448,498,446,880,1315,1587,690]. Two fascinating accounts of how the Enigma was broken are [735,796].

Further Reading

This is not a book about classical cryptography, so I will not dwell further on these subjects. Two excellent precomputer cryptology books are [587,1475]; [448] presents some modern cryptanalysis of cipher machines. Dorothy Denning discusses many of these ciphers in [456] and [880] has some fairly complex mathematical analysis of the same ciphers. Another older cryptography text, which discusses analog cryptography, is [99]. An article that presents a good overview of the subject is [579]. David Kahn's historical cryptography books are also excellent [794,795,796].

1.4 SIMPLE XOR

XOR is exclusive-or operation: '^' in C or $\oplus$ in mathematical notation. It's a standard operation on bits:

$$0 \oplus 0 = 0$$

$$0 \oplus 1 = 1$$

$$1 \oplus 0 = 1$$

$$1 \oplus 1 = 0$$

Also note that:

$$a \oplus a = 0$$

$$a \oplus b \oplus b = a$$

The simple-XOR algorithm is really an embarrassment; it's nothing more than a Vigenère polyalphabetic cipher. It's here only because of its prevalence in commercial software packages, at least those in the MS-DOS and Macintosh worlds [1502,1387]. Unfortunately, if a software security program proclaims that it has a "proprietary" encryption algorithm—significantly faster than DES—the odds are that it is some variant of this.

```
/* Usage:  crypto key input_file output_file */

void main (int argc, char *argv[])
{
    FILE *fi, *fo;
    char *cp;
    int c;

    if ((cp = argv[1]) && *cp!='\0') {
        if ((fi = fopen(argv[2], "rb")) != NULL) {
            if ((fo = fopen(argv[3], "wb")) != NULL) {
                while ((c = getc(fi)) != EOF) {
                    if (!*cp) cp = argv[1];
                    c ^= *(cp++);
                    putc(c,fo);
                }
                fclose(fo);
            }
            fclose(fi);
        }
    }
}
```

This is a symmetric algorithm. The plaintext is being XORed with a keyword to generate the ciphertext. Since XORing the same value twice restores the original, encryption and decryption use exactly the same program:

$$P \oplus K = C$$

$$C \oplus K = P$$

There's no real security here. This kind of encryption is trivial to break, even without computers [587,1475]. It will only take a few seconds with a computer.

Assume the plaintext is English. Furthermore, assume the key length is any small number of bytes. Here's how to break it:

1. Discover the length of the key by a procedure known as **counting coincidences** [577]. XOR the ciphertext against itself shifted various numbers of bytes, and count those bytes that are equal. If the displacement is a multiple of the key length, then something over 6 percent of the bytes will be equal. If it is not, then less than 0.4 percent will be equal (assuming a random key encrypting normal ASCII text; other plaintext will have different numbers). This is called the **index of coincidence**. The smallest displacement that indicates a multiple of the key length is the length of the key.

2. Shift the ciphertext by that length and XOR it with itself. This removes the key and leaves you with plaintext XORed with the plaintext shifted the length of the key. Since English has 1.3 bits of real information per byte (see Section 11.1), there is plenty of redundancy for determining a unique decryption.

Despite this, the list of software vendors that tout this toy algorithm as being “almost as secure as DES” is staggering [1387]. It is the algorithm (with a 160-bit repeated “key”) that the NSA finally allowed the U.S. digital cellular phone industry to use for voice privacy. An XOR might keep your kid sister from reading your files, but it won’t stop a cryptanalyst for more than a few minutes.

1.5 ONE-TIME PADS

Believe it or not, there is a perfect encryption scheme. It’s called a **one-time pad**, and was invented in 1917 by Major Joseph Mauborgne and AT&T’s Gilbert Vernam [794]. (Actually, a one-time pad is a special case of a threshold scheme; see Section 3.7.) Classically, a one-time pad is nothing more than a large nonrepeating set of truly random key letters, written on sheets of paper, and glued together in a pad. In its original form, it was a one-time tape for teletypewriters. The sender uses each key letter on the pad to encrypt exactly one plaintext character. Encryption is the addition modulo 26 of the plaintext character and the one-time pad key character.

Each key letter is used exactly once, for only one message. The sender encrypts the message and then destroys the used pages of the pad or used section of the tape. The receiver has an identical pad and uses each key on the pad, in turn, to decrypt each letter of the ciphertext. The receiver destroys the same pad pages or tape section after decrypting the message. New message—new key letters. For example, if the message is:

ONETIMEPAD

and the key sequence from the pad is

TBFRGFARFM

then the ciphertext is

IPKLPSFHGQ

because

$$O + T \bmod 26 = I$$

$$N + B \bmod 26 = P$$

$$E + F \bmod 26 = K$$

etc.

Assuming an eavesdropper can't get access to the one-time pad used to encrypt the message, this scheme is perfectly secure. A given ciphertext message is equally likely to correspond to any possible plaintext message of equal size.

Since every key sequence is equally likely (remember, the key letters are generated randomly), an adversary has no information with which to cryptanalyze the ciphertext. The key sequence could just as likely be:

POYYAEAAZX

which would decrypt to:

SALMONEGGS

or

BXFGBMTMXM

which would decrypt to:

GREENFLUID

This point bears repeating: Since every plaintext message is equally possible, there is no way for the cryptanalyst to determine which plaintext message is the correct one. A random key sequence added to a nonrandom plaintext message produces a completely random ciphertext message and no amount of computing power can change that.

The caveat, and this is a big one, is that the key letters have to be generated randomly. Any attacks against this scheme will be against the method used to generate the key letters. Using a pseudo-random number generator doesn't count; they always have nonrandom properties. If you use a real random source—this is much harder than it might first appear, see Section 17.14—it's secure.

The other important point is that you can never use the key sequence again, ever. Even if you use a multiple-gigabyte pad, if a cryptanalyst has multiple ciphertexts whose keys overlap, he can reconstruct the plaintext. He slides each pair of ciphertexts against each other and counts the number of matches at each position. If they are aligned right, the proportion of matches jumps suddenly—the exact percentages depend on the plaintext language. From this point cryptanalysis is easy. It's like the index of coincidence, but with just two "periods" to compare [904]. Don't do it.

The idea of a one-time pad can be easily extended to binary data. Instead of a one-time pad consisting of letters, use a one-time pad of bits. Instead of adding the plaintext to the one-time pad, use an XOR. To decrypt, XOR the ciphertext with the same one-time pad. Everything else remains the same and the security is just as perfect.

This all sounds good, but there are a few problems. Since the key bits must be random and can never be used again, the length of the key sequence must be equal to the length of the message. A one-time pad might be suitable for a few short messages, but it will never work for a 1.544 Mbps communications channel. You can store 650 megabytes worth of random bits on a CD-ROM, but there are problems. First, you want exactly two copies of the random bits, but CD-ROMs are economi-

cal only for large quantities. And second, you want to be able to destroy the bits already used. CD-ROM has no erase facilities except for physically destroying the entire disk. Digital tape is a much better medium for this sort of thing.

Even if you solve the key distribution and storage problem, you have to make sure the sender and receiver are perfectly synchronized. If the receiver is off by a bit (or if some bits are dropped during the transmission), the message won't make any sense. On the other hand, if some bits are altered during transmission (without any bits being added or removed—something far more likely to happen due to random noise), only those bits will be decrypted incorrectly. But on the other hand, a one-time pad provides no authenticity.

One-time pads have applications in today's world, primarily for ultra-secure low-bandwidth channels. The hotline between the United States and the former Soviet Union was (is it still active?) rumored to be encrypted with a one-time pad. Many Soviet spy messages to agents were encrypted using one-time pads. These messages are still secure today and will remain that way forever. It doesn't matter how long the supercomputers work on the problem. Even after the aliens from Andromeda land with their massive spaceships and undreamed-of computing power, they will not be able to read the Soviet spy messages encrypted with one-time pads (unless they can also go back in time and get the one-time pads).

1.6 COMPUTER ALGORITHMS

There are many cryptographic algorithms. These are three of the most common:

- DES (Data Encryption Standard) is the most popular computer encryption algorithm. DES is a U.S. and international standard. It is a symmetric algorithm; the same key is used for encryption and decryption.
- RSA (named for its creators—Rivest, Shamir, and Adleman) is the most popular public-key algorithm. It can be used for both encryption and digital signatures.
- DSA (Digital Signature Algorithm, used as part of the Digital Signature Standard) is another public-key algorithm. It cannot be used for encryption, but only for digital signatures.

These are the kinds of stuff this book is about.

1.7 LARGE NUMBERS

Throughout this book, I use various large numbers to describe different things in cryptography. Because it is so easy to lose sight of these numbers and what they signify, Table 1.1 gives physical analogues for some of them.

These numbers are order-of-magnitude estimates, and have been culled from a variety of sources. Many of the astrophysics numbers are explained in Freeman

TABLE 1.1
Large Numbers

Physical Analogue	Number
Odds of being killed by lightning (per day)	1 in 9 billion (2^{33})
Odds of winning the top prize in a U.S. state lottery	1 in 4,000,000 (2^{22})
Odds of winning the top prize in a U.S. state lottery and being killed by lightning in the same day	1 in 2^{61}
Odds of drowning (in the U.S. per year)	1 in 59,000 (2^{16})
Odds of being killed in an automobile accident (in the U.S. in 1993)	1 in 6100 (2^{13})
Odds of being killed in an automobile accident (in the U.S. per lifetime)	1 in 88 (2^7)
Time until the next ice age	14,000 (2^{14}) years
Time until the sun goes nova	10^9 (2^{30}) years
Age of the planet	10^9 (2^{30}) years
Age of the Universe	10^{10} (2^{34}) years
Number of atoms in the planet	10^{51} (2^{170})
Number of atoms in the sun	10^{57} (2^{190})
Number of atoms in the galaxy	10^{67} (2^{223})
Number of atoms in the Universe (dark matter excluded)	10^{77} (2^{265})
Volume of the Universe	10^{84} (2^{280}) cm^3
If the Universe is Closed:	
Total lifetime of the Universe	10^{11} (2^{37}) years
	10^{18} (2^{61}) seconds
If the Universe is Open:	
Time until low-mass stars cool off	10^{14} (2^{47}) years
Time until planets detach from stars	10^{15} (2^{50}) years
Time until stars detach from galaxies	10^{19} (2^{64}) years
Time until orbits decay by gravitational radiation	10^{20} (2^{67}) years
Time until black holes decay by the Hawking process	10^{64} (2^{213}) years
Time until all matter is liquid at zero temperature	10^{65} (2^{216}) years
Time until all matter decays to iron	$10^{10^{26}}$ years
Time until all matter collapses to black holes	$10^{10^{76}}$ years

Dyson's paper, "Time Without End: Physics and Biology in an Open Universe," in *Reviews of Modern Physics*, v. 52, n. 3, July 1979, pp. 447–460. Automobile accident deaths are calculated from the Department of Transportation's statistic of 163 deaths per million people in 1993 and an average lifespan of 69.7 years.

PART I



CRYPTOGRAPHIC PROTOCOLS

CHAPTER 2

Protocol Building Blocks

2.1 INTRODUCTION TO PROTOCOLS

The whole point of cryptography is to solve problems. (Actually, that's the whole point of computers—something many people tend to forget.) Cryptography solves problems that involve secrecy, authentication, integrity, and dishonest people. You can learn all about cryptographic algorithms and techniques, but these are academic unless they can solve a problem. This is why we are going to look at protocols first.

A **protocol** is a series of steps, involving two or more parties, designed to accomplish a task. This is an important definition. A “series of steps” means that the protocol has a sequence, from start to finish. Every step must be executed in turn, and no step can be taken before the previous step is finished. “Involving two or more parties” means that at least two people are required to complete the protocol; one person alone does not make a protocol. A person alone can perform a series of steps to accomplish a task (like baking a cake), but this is not a protocol. (Someone else must eat the cake to make it a protocol.) Finally, “designed to accomplish a task” means that the protocol must achieve something. Something that looks like a protocol but does not accomplish a task is not a protocol—it's a waste of time.

Protocols have other characteristics as well:

- Everyone involved in the protocol must know the protocol and all of the steps to follow in advance.
- Everyone involved in the protocol must agree to follow it.
- The protocol must be unambiguous; each step must be well defined and there must be no chance of a misunderstanding.
- The protocol must be complete; there must be a specified action for every possible situation.

The protocols in this book are organized as a series of steps. Execution of the protocol proceeds linearly through the steps, unless there are instructions to branch to another step. Each step involves at least one of two things: computations by one or more of the parties, or messages sent among the parties.

A **cryptographic protocol** is a protocol that uses cryptography. The parties can be friends and trust each other implicitly or they can be adversaries and not trust one another to give the correct time of day. A cryptographic protocol involves some cryptographic algorithm, but generally the goal of the protocol is something beyond simple secrecy. The parties participating in the protocol might want to share parts of their secrets to compute a value, jointly generate a random sequence, convince one another of their identity, or simultaneously sign a contract. The whole point of using cryptography in a protocol is to prevent or detect eavesdropping and cheating. If you have never seen these protocols before, they will radically change your ideas of what mutually distrustful parties can accomplish over a computer network. In general, this can be stated as:

- It should not be possible to do more or learn more than what is specified in the protocol.

This is a lot harder than it looks. In the next few chapters I discuss a lot of protocols. In some of them it is possible for one of the participants to cheat the other. In others, it is possible for an eavesdropper to subvert the protocol or learn secret information. Some protocols fail because the designers weren't thorough enough in their requirements definitions. Others fail because their designers weren't thorough enough in their analysis. Like algorithms, it is much easier to prove insecurity than it is to prove security.

The Purpose of Protocols

In daily life, there are informal protocols for almost everything: ordering goods over the telephone, playing poker, voting in an election. No one thinks much about these protocols; they have evolved over time, everyone knows how to use them, and they work reasonably well.

These days, more and more human interaction takes place over computer networks instead of face-to-face. Computers need formal protocols to do the same things that people do without thinking. If you moved from one state to another and found a voting booth that looked completely different from the ones you were used to, you could easily adapt. Computers are not nearly so flexible.

Many face-to-face protocols rely on people's presence to ensure fairness and security. Would you send a stranger a pile of cash to buy groceries for you? Would you play poker with someone if you couldn't see him shuffle and deal? Would you mail the government your secret ballot without some assurance of anonymity?

It is naïve to assume that people on computer networks are honest. It is naïve to assume that the managers of computer networks are honest. It is even naïve to assume that the designers of computer networks are honest. Most are, but the dis-

honest few can do a lot of damage. By formalizing protocols, we can examine ways in which dishonest parties can subvert them. Then we can develop protocols that are immune to that subversion.

In addition to formalizing behavior, protocols abstract the process of accomplishing a task from the mechanism by which the task is accomplished. A communications protocol is the same whether implemented on PCs or VAXs. We can examine the protocol without getting bogged down in the implementation details. When we are convinced we have a good protocol, we can implement it in everything from computers to telephones to intelligent muffin toasters.

The Players

To help demonstrate protocols, I have enlisted the aid of several people (see Table 2.1). Alice and Bob are the first two. They will perform all general two-person protocols. As a rule, Alice will initiate all protocols and Bob will respond. If the protocol requires a third or fourth person, Carol and Dave will perform those roles. Other actors will play specialized supporting roles; they will be introduced later.

Arbitrated Protocols

An **arbitrator** is a disinterested third party trusted to complete a protocol (see Figure 2.1a). Disinterested means that the arbitrator has no vested interest in the protocol and no particular allegiance to any of the parties involved. Trusted means that all people involved in the protocol accept what he says as true, what he does as correct, and that he will complete his part of the protocol. Arbitrators can help complete protocols between two mutually distrustful parties.

In the real world, lawyers are often used as arbitrators. For example, Alice is selling a car to Bob, a stranger. Bob wants to pay by check, but Alice has no way of knowing if the check is good. Alice wants the check to clear before she turns the title over to Bob. Bob, who doesn't trust Alice any more than she trusts him, doesn't want to hand over a check without receiving a title.

TABLE 2.1
Dramatis Personae

Alice	First participant in all the protocols
Bob	Second participant in all the protocols
Carol	Participant in the three- and four-party protocols
Dave	Participant in the four-party protocols
Eve	Eavesdropper
Mallory	Malicious active attacker
Trent	Trusted arbitrator
Walter	Warden; he'll be guarding Alice and Bob in some protocols
Peggy	Prover
Victor	Verifier

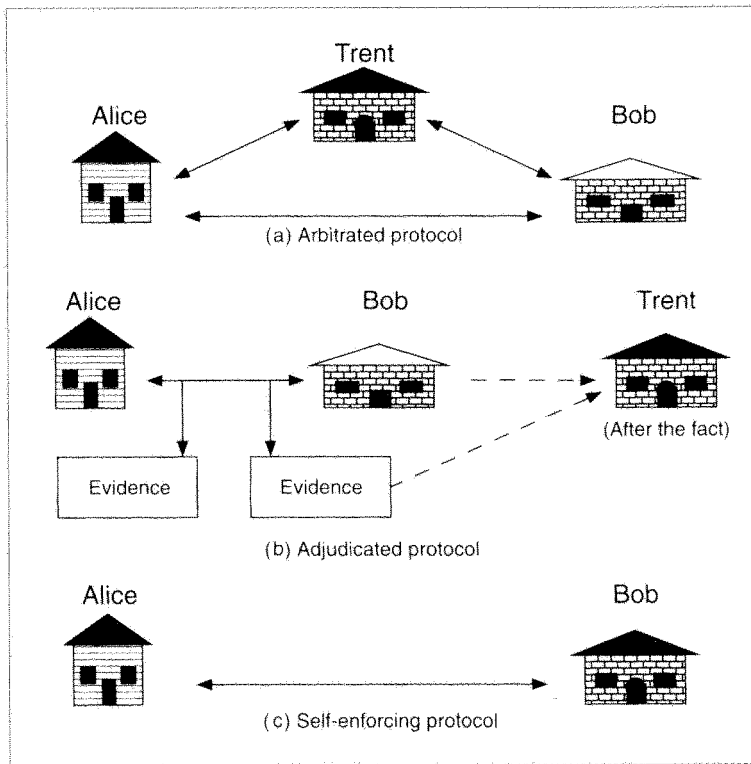


Figure 2.1 Types of protocols.

Enter a lawyer trusted by both. With his help, Alice and Bob can use the following protocol to ensure that neither cheats the other:

- (1) Alice gives the title to the lawyer.
- (2) Bob gives the check to Alice.
- (3) Alice deposits the check.
- (4) After waiting a specified time period for the check to clear, the lawyer gives the title to Bob. If the check does not clear within the specified time period, Alice shows proof of this to the lawyer and the lawyer returns the title to Alice.

In this protocol, Alice trusts the lawyer not to give Bob the title unless the check has cleared, and to give it back to her if the check does not clear. Bob trusts the lawyer to hold the title until the check clears, and to give it to him once it does. The lawyer doesn't care if the check clears. He will do his part of the protocol in either case, because he will be paid in either case.

In the example, the lawyer is playing the part of an escrow agent. Lawyers also act as arbitrators for wills and sometimes for contract negotiations. The various stock exchanges act as arbitrators between buyers and sellers.

Bankers also arbitrate protocols. Bob can use a certified check to buy a car from Alice:

- (1) Bob writes a check and gives it to the bank.
- (2) After putting enough of Bob's money on hold to cover the check, the bank certifies the check and gives it back to Bob.
- (3) Alice gives the title to Bob and Bob gives the certified check to Alice.
- (4) Alice deposits the check.

This protocol works because Alice trusts the banker's certification. Alice trusts the bank to hold Bob's money for her, and not to use it to finance shaky real estate operations in mosquito-infested countries.

A notary public is another arbitrator. When Bob receives a notarized document from Alice, he is convinced that Alice signed the document voluntarily and with her own hand. The notary can, if necessary, stand up in court and attest to that fact.

The concept of an arbitrator is as old as society. There have always been people—rulers, priests, and so on—who have the authority to act fairly. Arbitrators have a certain social role and position in our society; betraying the public trust would jeopardize that. Lawyers who play games with escrow accounts face almost-certain disbarment, for example. This picture of trust doesn't always exist in the real world, but it's the ideal.

This ideal can translate to the computer world, but there are several problems with computer arbitrators:

- It is easier to find and trust a neutral third party if you know who the party is and can see his face. Two parties suspicious of each other are also likely to be suspicious of a faceless arbitrator somewhere else on the network.
- The computer network must bear the cost of maintaining an arbitrator. We all know what lawyers charge; who wants to bear that kind of network overhead?
- There is a delay inherent in any arbitrated protocol.
- The arbitrator must deal with every transaction; he is a bottleneck in large-scale implementations of any protocol. Increasing the number of arbitrators in the implementation can mitigate this problem, but that increases the cost.
- Since everyone on the network must trust the arbitrator, he represents a vulnerable point for anyone trying to subvert the network.

Even so, arbitrators still have a role to play. In protocols using a trusted arbitrator, the part will be played by Trent.

Adjudicated Protocols

Because of the high cost of hiring arbitrators, arbitrated protocols can be subdivided into two lower-level **subprotocols**. One is a nonarbitrated subprotocol, executed every time parties want to complete the protocol. The other is an arbitrated subprotocol, executed only in exceptional circumstances—when there is a dispute. This special type of arbitrator is called an **adjudicator** (see Figure 2.1b).

An adjudicator is also a disinterested and trusted third party. Unlike an arbitrator, he is not directly involved in every protocol. The adjudicator is called in only to determine whether a protocol was performed fairly.

Judges are professional adjudicators. Unlike a notary public, a judge is brought in only if there is a dispute. Alice and Bob can enter into a contract without a judge. A judge never sees the contract until one of them hauls the other into court.

This contract-signing protocol can be formalized in this way:

Nonarbitrated subprotocol (executed every time):

- (1) Alice and Bob negotiate the terms of the contract.
- (2) Alice signs the contract.
- (3) Bob signs the contract.

Adjudicated subprotocol (executed only in case of a dispute):

- (4) Alice and Bob appear before a judge.
- (5) Alice presents her evidence.
- (6) Bob presents his evidence.
- (7) The judge rules on the evidence.

The difference between an adjudicator and an arbitrator (as used in this book) is that the adjudicator is not always necessary. In a dispute, a judge is called in to adjudicate. If there is no dispute, using a judge is unnecessary.

There are adjudicated computer protocols. These protocols rely on the parties to be honest; but if someone suspects cheating, a body of data exists so that a trusted third party could determine if someone cheated. In a good adjudicated protocol, the adjudicator could also determine the cheater's identity. Instead of preventing cheating, adjudicated protocols detect cheating. The inevitability of detection acts as a preventive and discourages cheating.

Self-Enforcing Protocols

A **self-enforcing protocol** is the best type of protocol. The protocol itself guarantees fairness (see Figure 2.1c). No arbitrator is required to complete the protocol. No adjudicator is required to resolve disputes. The protocol is constructed so that there

cannot be any disputes. If one of the parties tries to cheat, the other party immediately detects the cheating and the protocol stops. Whatever the cheating party hoped would happen by cheating, doesn't happen.

In the best of all possible worlds, every protocol would be self-enforcing. Unfortunately, there is not a self-enforcing protocol for every situation.

Attacks against Protocols

Cryptographic attacks can be directed against the cryptographic algorithms used in protocols, against the cryptographic techniques used to implement the algorithms and protocols, or against the protocols themselves. Since this section of the book discusses protocols, I will assume that the cryptographic algorithms and techniques are secure. I will only examine attacks against the protocols.

People can try various ways to attack a protocol. Someone not involved in the protocol can eavesdrop on some or all of the protocol. This is called a **passive attack**, because the attacker does not affect the protocol. All he can do is observe the protocol and attempt to gain information. This kind of attack corresponds to a ciphertext-only attack, as discussed in Section 1.1. Since passive attacks are difficult to detect, protocols try to prevent passive attacks rather than detect them. In these protocols, the part of the eavesdropper will be played by Eve.

Alternatively, an attacker could try to alter the protocol to his own advantage. He could pretend to be someone else, introduce new messages in the protocol, delete existing messages, substitute one message for another, replay old messages, interrupt a communications channel, or alter stored information in a computer. These are called **active attacks**, because they require active intervention. The form of these attacks depends on the network.

Passive attackers try to gain information about the parties involved in the protocol. They collect messages passing among various parties and attempt to cryptanalyze them. Active attacks, on the other hand, can have much more diverse objectives. The attacker could be interested in obtaining information, degrading system performance, corrupting existing information, or gaining unauthorized access to resources.

Active attacks are much more serious, especially in protocols in which the different parties don't necessarily trust one another. The attacker does not have to be a complete outsider. He could be a legitimate system user. He could be the system administrator. There could even be many active attackers working together. Here, the part of the malicious active attacker will be played by Mallory.

It is also possible that the attacker could be one of the parties involved in the protocol. He may lie during the protocol or not follow the protocol at all. This type of attacker is called a **cheater**. **Passive cheaters** follow the protocol, but try to obtain more information than the protocol intends them to. **Active cheaters** disrupt the protocol in progress in an attempt to cheat.

It is very difficult to maintain a protocol's security if most of the parties involved are active cheaters, but sometimes it is possible for legitimate parties to detect that active cheating is going on. Certainly, protocols should be secure against passive cheating.

2.2 COMMUNICATIONS USING SYMMETRIC CRYPTOGRAPHY

How do two parties communicate securely? They encrypt their communications, of course. The complete protocol is more complicated than that. Let's look at what must happen for Alice to send an encrypted message to Bob.

- (1) Alice and Bob agree on a cryptosystem.
- (2) Alice and Bob agree on a key.
- (3) Alice takes her plaintext message and encrypts it using the encryption algorithm and the key. This creates a ciphertext message.
- (4) Alice sends the ciphertext message to Bob.
- (5) Bob decrypts the ciphertext message with the same algorithm and key and reads it.

What can Eve, sitting between Alice and Bob, learn from listening in on this protocol? If all she hears is the transmission in step (4), she must try to cryptanalyze the ciphertext. This passive attack is a ciphertext-only attack; we have algorithms that are resistant (as far as we know) to whatever computing power Eve could realistically bring to bear on the problem.

Eve isn't stupid, though. She also wants to listen in on steps (1) and (2). Then, she would know the algorithm and the key—just as well as Bob. When the message comes across the communications channel in step (4), all she has to do is decrypt it herself.

A good cryptosystem is one in which all the security is inherent in knowledge of the key and none is inherent in knowledge of the algorithm. This is why key management is so important in cryptography. With a symmetric algorithm, Alice and Bob can perform step (1) in public, but they must perform step (2) in secret. The key must remain secret before, during, and after the protocol—as long as the message must remain secret—otherwise the message will no longer be secure. (Public-key cryptography solves this problem another way, and will be discussed in Section 2.5.)

Mallory, an active attacker, could do a few other things. He could attempt to break the communications path in step (4), ensuring that Alice could not talk to Bob at all. Mallory could also intercept Alice's messages and substitute his own. If he knew the key (by intercepting the communication in step (2), or by breaking the cryptosystem), he could encrypt his own message and send it to Bob in place of the intercepted message. Bob would have no way of knowing that the message had not come from Alice. If Mallory didn't know the key, he could only create a replacement message that would decrypt to gibberish. Bob, thinking the message came from Alice, might conclude that either the network or Alice had some serious problems.

What about Alice? What can she do to disrupt the protocol? She can give a copy of the key to Eve. Now Eve can read whatever Bob says. She can reprint his words in *The New York Times*. Although serious, this is not a problem with the protocol. There is nothing to stop Alice from giving Eve a copy of the plaintext at any point

during the protocol. Of course, Bob could also do anything that Alice could. This protocol assumes that Alice and Bob trust each other.

In summary, symmetric cryptosystems have the following problems:

- Keys must be distributed in secret. They are as valuable as all the messages they encrypt, since knowledge of the key gives knowledge of all the messages. For encryption systems that span the world, this can be a daunting task. Often couriers hand-carry keys to their destinations.
- If a key is compromised (stolen, guessed, extorted, bribed, etc.), then Eve can decrypt all message traffic encrypted with that key. She can also pretend to be one of the parties and produce false messages to fool the other party.
- Assuming a separate key is used for each pair of users in a network, the total number of keys increases rapidly as the number of users increases. A network of n users requires $n(n-1)/2$ keys. For example, 10 users require 45 different keys to talk with one another and 100 users require 4950 keys. This problem can be minimized by keeping the number of users small, but that is not always possible.

2.3 ONE-WAY FUNCTIONS

The notion of a **one-way function** is central to public-key cryptography. While not protocols in themselves, one-way functions are a fundamental building block for most of the protocols discussed in this book.

One-way functions are relatively easy to compute, but significantly harder to reverse. That is, given x it is easy to compute $f(x)$, but given $f(x)$ it is hard to compute x . In this context, “hard” is defined as something like: It would take millions of years to compute x from $f(x)$, even if all the computers in the world were assigned to the problem.

Breaking a plate is a good example of a one-way function. It is easy to smash a plate into a thousand tiny pieces. However, it's not easy to put all of those tiny pieces back together into a plate.

This sounds good, but it's a lot of smoke and mirrors. If we are being strictly mathematical, we have no proof that one-way functions exist, nor any real evidence that they can be constructed [230,530,600,661]. Even so, many functions look and smell one-way: We can compute them efficiently and, as of yet, know of no easy way to reverse them. For example, in a finite field x^2 is easy to compute, but $x^{1/2}$ is much harder. For the rest of this section, I'm going to pretend that there are one-way functions. I'll talk more about this in Section 11.2.

So, what good are one-way functions? We can't use them for encryption as is. A message encrypted with the one-way function isn't useful; no one could decrypt it. (Exercise: Write a message on a plate, smash the plate into tiny bits, and then give the bits to a friend. Ask your friend to read the message. Observe how impressed

he is with the one-way function.) For public-key cryptography, we need something else (although there are cryptographic applications for one-way functions—see Section 3.2).

A **trapdoor one-way function** is a special type of one-way function, one with a secret trapdoor. It is easy to compute in one direction and hard to compute in the other direction. But, if you know the secret, you can easily compute the function in the other direction. That is, it is easy to compute $f(x)$ given x , and hard to compute x given $f(x)$. However, there is some secret information, y , such that given $f(x)$ and y it is easy to compute x .

Taking a watch apart is a good example of a trap-door one-way function. It is easy to disassemble a watch into hundreds of minuscule pieces. It is very difficult to put those tiny pieces back together into a working watch. However, with the secret information—the assembly instructions of the watch—it is much easier to put the watch back together.

2.4 ONE-WAY HASH FUNCTIONS

A **one-way hash function** has many names: compression function, contraction function, message digest, fingerprint, cryptographic checksum, message integrity check (MIC), and manipulation detection code (MDC). Whatever you call it, it is central to modern cryptography. One-way hash functions are another building block for many protocols.

Hash functions have been used in computer science for a long time. A hash function is a function, mathematical or otherwise, that takes a variable-length input string (called a **pre-image**) and converts it to a fixed-length (generally smaller) output string (called a **hash value**). A simple hash function would be a function that takes pre-image and returns a byte consisting of the XOR of all the input bytes.

The point here is to fingerprint the pre-image: to produce a value that indicates whether a candidate pre-image is likely to be the same as the real pre-image. Because hash functions are typically many-to-one, we cannot use them to determine with certainty that the two strings are equal, but we can use them to get a reasonable assurance of accuracy.

A one-way hash function is a hash function that works in one direction: It is easy to compute a hash value from pre-image, but it is hard to generate a pre-image that hashes to a particular value. The hash function previously mentioned is not one-way: Given a particular byte value, it is trivial to generate a string of bytes whose XOR is that value. You can't do that with a one-way hash function. A good one-way hash function is also **collision-free**: It is hard to generate two pre-images with the same hash value.

The hash function is public; there's no secrecy to the process. The security of a one-way hash function is its one-wayness. The output is not dependent on the input in any discernible way. A single bit change in the pre-image changes, on the average, half of the bits in the hash value. Given a hash value, it is computationally unfeasible to find a pre-image that hashes to that value.

Think of it as a way of fingerprinting files. If you want to verify that someone has a particular file (that you also have), but you don't want him to send it to you, then ask him for the hash value. If he sends you the correct hash value, then it is almost certain that he has that file. This is particularly useful in financial transactions, where you don't want a withdrawal of \$100 to turn into a withdrawal of \$1000 somewhere in the network. Normally, you would use a one-way hash function without a key, so that anyone can verify the hash. If you want only the recipient to be able to verify the hash, then read the next section.

Message Authentication Codes

A **message authentication code (MAC)**, also known as a data authentication code (DAC), is a one-way hash function with the addition of a secret key (see Section 18.14). The hash value is a function of both the pre-image and the key. The theory is exactly the same as hash functions, except only someone with the key can verify the hash value. You can create a MAC out of a hash function or a block encryption algorithm; there are also dedicated MACs.

2.5 COMMUNICATIONS USING PUBLIC-KEY CRYPTOGRAPHY

Think of a symmetric algorithm as a safe. The key is the combination. Someone with the combination can open the safe, put a document inside, and close it again. Someone else with the combination can open the safe and take the document out. Anyone without the combination is forced to learn safecracking.

In 1976, Whitfield Diffie and Martin Hellman changed that paradigm of cryptography forever [496]. (The NSA has claimed knowledge of the concept as early as 1966, but has offered no proof.) They described **public-key cryptography**. They used two different keys—one public and the other private. It is computationally hard to deduce the private key from the public key. Anyone with the public key can encrypt a message but not decrypt it. Only the person with the private key can decrypt the message. It is as if someone turned the cryptographic safe into a mailbox. Putting mail in the mailbox is analogous to encrypting with the public key; anyone can do it. Just open the slot and drop it in. Getting mail out of a mailbox is analogous to decrypting with the private key. Generally it's hard; you need welding torches. However, if you have the secret (the physical key to the mailbox), it's easy to get mail out of a mailbox.

Mathematically, the process is based on the trap-door one-way functions previously discussed. Encryption is the easy direction. Instructions for encryption are the public key; anyone can encrypt a message. Decryption is the hard direction. It's made hard enough that people with Cray computers and thousands (even millions) of years couldn't decrypt the message without the secret. The secret, or trapdoor, is the private key. With that secret, decryption is as easy as encryption.

This is how Alice can send a message to Bob using public-key cryptography:

- (1) Alice and Bob agree on a public-key cryptosystem.

- (2) Bob sends Alice his public key.
- (3) Alice encrypts her message using Bob's public key and sends it to Bob.
- (4) Bob decrypts Alice's message using his private key.

Notice how public-key cryptography solves the key-management problem with symmetric cryptosystems. Before, Alice and Bob had to agree on a key in secret. Alice could choose one at random, but she still had to get it to Bob. She could hand it to him sometime beforehand, but that requires foresight. She could send it to him by secure courier, but that takes time. Public-key cryptography makes it easy. With no prior arrangements, Alice can send a secure message to Bob. Eve, listening in on the entire exchange, has Bob's public key and a message encrypted in that key, but cannot recover either Bob's private key or the message.

More commonly, a network of users agrees on a public-key cryptosystem. Every user has his or her own public key and private key, and the public keys are all published in a database somewhere. Now the protocol is even easier:

- (1) Alice gets Bob's public key from the database.
- (2) Alice encrypts her message using Bob's public key and sends it to Bob.
- (3) Bob then decrypts Alice's message using his private key.

In the first protocol, Bob had to send Alice his public key before she could send him a message. The second protocol is more like traditional mail. Bob is not involved in the protocol until he wants to read his message.

Hybrid Cryptosystems

The first public-key algorithms became public at the same time that DES was being discussed as a proposed standard. This resulted in some partisan politics in the cryptographic community. As Diffie described it [494]:

The excitement public key cryptosystems provoked in the popular and scientific press was not matched by corresponding acceptance in the cryptographic establishment, however. In the same year that public key cryptography was discovered, the National Security Agency (NSA), proposed a conventional cryptographic system, designed by International Business Machines (IBM), as a federal *Data Encryption Standard* (DES). Marty Hellman and I criticized the proposal on the ground that its key was too small, but manufacturers were gearing up to support the proposed standard and our criticism was seen by many as an attempt to disrupt the standards-making process to the advantage of our own work. Public key cryptography in its turn was attacked, in sales literature [1125] and technical papers [849, 1159] alike, more as though it were a competing product than a recent research discovery. This, however, did not deter the NSA from claiming its share of the credit. Its director, in the words of the *Encyclopedia Britannica* [1461], pointed out that "two-key cryptography had been discovered at the agency a decade earlier," although no evidence for this claim was ever offered publicly.

In the real world, public-key algorithms are not a substitute for symmetric algorithms. They are not used to encrypt messages; they are used to encrypt keys. There are two reasons for this:

1. Public-key algorithms are slow. Symmetric algorithms are generally at least 1000 times faster than public-key algorithms. Yes, computers are getting faster and faster, and in 15 years computers will be able to do public-key cryptography at speeds comparable to symmetric cryptography today. But bandwidth requirements are also increasing, and there will always be the need to encrypt data faster than public-key cryptography can manage.
2. Public-key cryptosystems are vulnerable to chosen-plaintext attacks. If $C = E(P)$, when P is one plaintext out of a set of n possible plaintexts, then a cryptanalyst only has to encrypt all n possible plaintexts and compare the results with C (remember, the encryption key is public). He won't be able to recover the decryption key this way, but he will be able to determine P .

A chosen-plaintext attack can be particularly effective if there are relatively few possible encrypted messages. For example, if P were a dollar amount less than \$1,000,000, this attack would work; the cryptanalyst tries all million possible dollar amounts. (Probabilistic encryption solves the problem; see Section 23.15.) Even if P is not as well-defined, this attack can be very effective. Simply knowing that a ciphertext does not correspond to a particular plaintext can be useful information. Symmetric cryptosystems are not vulnerable to this attack because a cryptanalyst cannot perform trial encryptions with an unknown key.

In most practical implementations public-key cryptography is used to secure and distribute **session keys**; those session keys are used with symmetric algorithms to secure message traffic [879]. This is sometimes called a **hybrid cryptosystem**.

- (1) Bob sends Alice his public key.
- (2) Alice generates a random session key, K , encrypts it using Bob's public key, and sends it to Bob.

$$E_B(K)$$

- (3) Bob decrypts Alice's message using his private key to recover the session key.

$$D_B(E_B(K)) = K$$

- (4) Both of them encrypt their communications using the same session key.

Using public-key cryptography for key distribution solves a very important key-management problem. With symmetric cryptography, the data encryption key sits around until it is used. If Eve ever gets her hands on it, she can decrypt messages encrypted with it. With the previous protocol, the session key is created when it is needed to encrypt communications and destroyed when it is no longer needed. This drastically reduces the risk of compromising the session key. Of course, the private

key is vulnerable to compromise, but it is at less risk because it is only used once per communication to encrypt a session key. This is further discussed in Section 3.1.

Merkle's Puzzles

Ralph Merkle invented the first construction of public-key cryptography. In 1974 he registered for a course in computer security at the University of California, Berkeley, taught by Lance Hoffman. His term paper topic, submitted early in the term, addressed the problem of "Secure Communication over Insecure Channels" [1064]. Hoffman could not understand Merkle's proposal and eventually Merkle dropped the course. He continued to work on the problem, despite continuing failure to make his results understood.

Merkle's technique was based on "puzzles" that were easier to solve for the sender and receiver than for an eavesdropper. Here's how Alice sends an encrypted message to Bob without first having to exchange a key with him.

- (1) Bob generates 2^{20} , or about a million, messages of the form: "This is puzzle number x . This is the secret key number y ," where x is a random number and y is a random secret key. Both x and y are different for each message. Using a symmetric algorithm, he encrypts each message with a different 20-bit key and sends them all to Alice.
- (2) Alice chooses one message at random and performs a brute-force attack to recover the plaintext. This is a large, but not impossible, amount of work.
- (3) Alice encrypts her secret message with the key she recovered and some symmetric algorithm, and sends it to Bob along with x .
- (4) Bob knows which secret key y he encrypts in message x , so he can decrypt the message.

Eve can break this system, but she has to do far more work than either Alice or Bob. To recover the message in step (3), she has to perform a brute-force attack against each of Bob's 2^{20} messages in step (1); this attack has a complexity of 2^{40} . The x values won't help Eve either; they were assigned randomly in step (1). In general, Eve has to expend approximately the square of the effort that Alice expends.

This n to n^2 advantage is small by cryptographic standards, but in some circumstances it may be enough. If Alice and Bob can try ten thousand keys per second, it will take them a minute each to perform their steps and another minute to communicate the puzzles from Bob to Alice on a 1.544 MB link. If Eve had comparable computing facilities, it would take her about a year to break the system. Other algorithms are even harder to break.

2.6 DIGITAL SIGNATURES

Handwritten signatures have long been used as proof of authorship of, or at least agreement with, the contents of a document. What is it about a signature that is so compelling [1392]?

1. The signature is authentic. The signature convinces the document's recipient that the signer deliberately signed the document.
2. The signature is unforgeable. The signature is proof that the signer, and no one else, deliberately signed the document.
3. The signature is not reusable. The signature is part of the document; an unscrupulous person cannot move the signature to a different document.
4. The signed document is unalterable. After the document is signed, it cannot be altered.
5. The signature cannot be repudiated. The signature and the document are physical things. The signer cannot later claim that he or she didn't sign it.

In reality, none of these statements about signatures is completely true. Signatures can be forged, signatures can be lifted from one piece of paper and moved to another, and documents can be altered after signing. However, we are willing to live with these problems because of the difficulty in cheating and the risk of detection.

We would like to do this sort of thing on computers, but there are problems. First, computer files are trivial to copy. Even if a person's signature were difficult to forge (a graphical image of a written signature, for example), it would be easy to cut and paste a valid signature from one document to another document. The mere presence of such a signature means nothing. Second, computer files are easy to modify after they are signed, without leaving any evidence of modification.

Signing Documents with Symmetric Cryptosystems and an Arbitrator

Alice wants to sign a digital message and send it to Bob. With the help of Trent and a symmetric cryptosystem, she can.

Trent is a powerful, trusted arbitrator. He can communicate with both Alice and Bob (and everyone else who may want to sign a digital document). He shares a secret key, K_A , with Alice, and a different secret key, K_B , with Bob. These keys have been established long before the protocol begins and can be reused multiple times for multiple signings.

- (1) Alice encrypts her message to Bob with K_A and sends it to Trent.
- (2) Trent decrypts the message with K_A .
- (3) Trent takes the decrypted message and a statement that he has received this message from Alice, and encrypts the whole bundle with K_B .
- (4) Trent sends the encrypted bundle to Bob.
- (5) Bob decrypts the bundle with K_B . He can now read both the message and Trent's certification that Alice sent it.

How does Trent know that the message is from Alice and not from some imposter? He infers it from the message's encryption. Since only he and Alice share their secret key, only Alice could encrypt a message using it.

Is this as good as a paper signature? Let's look at the characteristics we want:

1. This signature is authentic. Trent is a trusted arbitrator and Trent knows that the message came from Alice. Trent's certification serves as proof to Bob.
2. This signature is unforgeable. Only Alice (and Trent, but everyone trusts him) knows K_A , so only Alice could have sent Trent a message encrypted with K_A . If someone tried to impersonate Alice, Trent would have immediately realized this in step (2) and would not certify its authenticity.
3. This signature is not reusable. If Bob tried to take Trent's certification and attach it to another message, Alice would cry foul. An arbitrator (it could be Trent or it could be a completely different arbitrator with access to the same information) would ask Bob to produce both the message and Alice's encrypted message. The arbitrator would then encrypt the message with K_A and see that it did not match the encrypted message that Bob gave him. Bob, of course, could not produce an encrypted message that matches because he does not know K_A .
4. The signed document is unalterable. Were Bob to try to alter the document after receipt, Trent could prove foul play in exactly the same manner just described.
5. The signature cannot be repudiated. Even if Alice later claims that she never sent the message, Trent's certification says otherwise. Remember, Trent is trusted by everyone; what he says is true.

If Bob wants to show Carol a document signed by Alice, he can't reveal his secret key to her. He has to go through Trent again:

- (1) Bob takes the message and Trent's statement that the message came from Alice, encrypts them with K_B , and sends them back to Trent.
- (2) Trent decrypts the bundle with K_B .
- (3) Trent checks his database and confirms that the original message came from Alice.
- (4) Trent re-encrypts the bundle with the secret key he shares with Carol, K_C , and sends it to Carol.
- (5) Carol decrypts the bundle with K_C . She can now read both the message and Trent's certification that Alice sent it.

These protocols work, but they're time-consuming for Trent. He must spend his days decrypting and encrypting messages, acting as the intermediary between every pair of people who want to send signed documents to one another. He must keep a database of messages (although this can be avoided by sending the recipient a copy of the sender's encrypted message). He is a bottleneck in any communications system, even if he's a mindless software program.

Harder still is creating and maintaining someone like Trent, someone that everyone on the network trusts. Trent has to be infallible; if he makes even one mistake in a million signatures, no one is going to trust him. Trent has to be completely secure. If his database of secret keys ever got out or if someone managed to modify his programming, everyone's signatures would be completely useless. False documents purported to be signed years ago could appear. Chaos would result. Governments would collapse. Anarchy would reign. This might work in theory, but it doesn't work very well in practice.

Digital Signature Trees

Ralph Merkle proposed a digital signature scheme based on secret-key cryptography, producing an infinite number of one-time signatures using a tree structure [1067,1068]. The basic idea of this scheme is to place the root of the tree in some public file, thereby authenticating it. The root signs one message and authenticates its sub-nodes in the tree. Each of these nodes signs one message and authenticates its sub-nodes, and so on.

Signing Documents with Public-Key Cryptography

There are public-key algorithms that can be used for digital signatures. In some algorithms—RSA is an example (see Section 19.3)—either the public key or the private key can be used for encryption. Encrypt a document using your private key, and you have a secure digital signature. In other cases—DSA is an example (see Section 20.1)—there is a separate algorithm for digital signatures that cannot be used for encryption. This idea was first invented by Diffie and Hellman [496] and further expanded and elaborated on in other texts [1282,1328,1024,1283,426]. See [1099] for a good survey of the field.

The basic protocol is simple:

- (1) Alice encrypts the document with her private key, thereby signing the document.
- (2) Alice sends the signed document to Bob.
- (3) Bob decrypts the document with Alice's public key, thereby verifying the signature.

This protocol is far better than the previous one. Trent is not needed to either sign or verify signatures. (He is needed to certify that Alice's public key is indeed her public key.) The parties do not even need Trent to resolve disputes: If Bob cannot perform step (3), then he knows the signature is not valid.

This protocol also satisfies the characteristics we're looking for:

1. The signature is authentic; when Bob verifies the message with Alice's public key, he knows that she signed it.
2. The signature is unforgeable; only Alice knows her private key.
3. The signature is not reusable; the signature is a function of the document and cannot be transferred to any other document.

4. The signed document is unalterable; if there is any alteration to the document, the signature can no longer be verified with Alice's public key.
5. The signature cannot be repudiated. Bob doesn't need Alice's help to verify her signature.

Signing Documents and Timestamps

Actually, Bob can cheat Alice in certain circumstances. He can reuse the document and signature together. This is no problem if Alice signed a contract (what's another copy of the same contract, more or less?), but it can be very exciting if Alice signed a digital check.

Let's say Alice sends Bob a signed digital check for \$100. Bob takes the check to the bank, which verifies the signature and moves the money from one account to the other. Bob, who is an unscrupulous character, saves a copy of the digital check. The following week, he again takes it to the bank (or maybe to a different bank). The bank verifies the signature and moves the money from one account to the other. If Alice never balances her checkbook, Bob can keep this up for years.

Consequently, digital signatures often include timestamps. The date and time of the signature are attached to the message and signed along with the rest of the message. The bank stores this timestamp in a database. Now, when Bob tries to cash Alice's check a second time, the bank checks the timestamp against its database. Since the bank already cashed a check from Alice with the same timestamp, the bank calls the police. Bob then spends 15 years in Leavenworth prison reading up on cryptographic protocols.

Signing Documents with Public-Key Cryptography and One-Way Hash Functions

In practical implementations, public-key algorithms are often too inefficient to sign long documents. To save time, digital signature protocols are often implemented with one-way hash functions [432,433]. Instead of signing a document, Alice signs the hash of the document. In this protocol, both the one-way hash function and the digital signature algorithm are agreed upon beforehand.

- (1) Alice produces a one-way hash of a document.
- (2) Alice encrypts the hash with her private key, thereby signing the document.
- (3) Alice sends the document and the signed hash to Bob.
- (4) Bob produces a one-way hash of the document that Alice sent. He then, using the digital signature algorithm, decrypts the signed hash with Alice's public key. If the signed hash matches the hash he generated, the signature is valid.

Speed increases drastically and, since the chances of two different documents having the same 160-bit hash are only one in 2^{160} , anyone can safely equate a signature of the hash with a signature of the document. If a non-one-way hash function were

used, it would be an easy matter to create multiple documents that hashed to the same value, so that anyone signing a particular document would be duped into signing a multitude of documents.

This protocol has other benefits. First, the signature can be kept separate from the document. Second, the recipient's storage requirements for the document and signature are much smaller. An archival system can use this type of protocol to verify the existence of documents without storing their contents. The central database could just store the hashes of files. It doesn't have to see the files at all; users submit their hashes to the database, and the database timestamps the submissions and stores them. If there is any disagreement in the future about who created a document and when, the database could resolve it by finding the hash in its files. This system has vast implications concerning privacy: Alice could copyright a document but still keep the document secret. Only if she wished to prove her copyright would she have to make the document public. (See Section 4.1).

Algorithms and Terminology

There are many digital signature algorithms. All of them are public-key algorithms with secret information to sign documents and public information to verify signatures. Sometimes the signing process is called **encrypting with a private key** and the verification process is called **decrypting with a public key**. This is misleading and is only true for one algorithm, RSA. And different algorithms have different implementations. For example, one-way hash functions and timestamps sometimes add extra steps to the process of signing and verifying. Many algorithms can be used for digital signatures, but not for encryption.

In general, I will refer to the signing and verifying processes without any details of the algorithms involved. Signing a message with private key K is:

$$S_K(M)$$

and verifying a signature with the corresponding public key is:

$$V_K(M)$$

The bit string attached to the document when signed (in the previous example, the one-way hash of the document encrypted with the private key) will be called the **digital signature**, or just the **signature**. The entire protocol, by which the receiver of a message is convinced of the identity of the sender and the integrity of the message, is called authentication. Further details on these protocols are in Section 3.2.

Multiple Signatures

How could Alice and Bob sign the same digital document? Without one-way hash functions, there are two options. One is that Alice and Bob sign separate copies of the document itself. The resultant message would be over twice the size of the original document. The second is that Alice signs the document first and then Bob signs Alice's signature. This works, but it is impossible to verify Alice's signature without also verifying Bob's.

With one-way hash functions, multiple signatures are easy:

- (1) Alice signs the hash of the document.
- (2) Bob signs the hash of the document.
- (3) Bob sends his signature to Alice.
- (4) Alice sends the document, her signature, and Bob's signature to Carol.
- (5) Carol verifies both Alice's signature and Bob's signature.

Alice and Bob can do steps (1) and (2) either in parallel or in series. In step (5), Carol can verify one signature without having to verify the other.

Nonrepudiation and Digital Signatures

Alice can cheat with digital signatures and there's nothing that can be done about it. She can sign a document and then later claim that she did not. First, she signs the document normally. Then, she anonymously publishes her private key, conveniently loses it in a public place, or just pretends to do either one. Alice then claims that her signature has been compromised and that others are using it, pretending to be her. She disavows signing the document and any others that she signed using that private key. This is called repudiation.

Timestamps can limit the effects of this kind of cheating, but Alice can always claim that her key was compromised earlier. If Alice times things well, she can sign a document and then successfully claim that she didn't. This is why there is so much talk about private keys buried in tamper-resistant modules—so that Alice can't get at hers and abuse it.

Although nothing can be done about this possible abuse, one can take steps to guarantee that old signatures are not invalidated by actions taken in disputing new ones. (For example, Alice could "lose" her key to keep from paying Bob for the junk car he sold her yesterday and, in the process, invalidate her bank account.) The solution is for the receiver of a signed document to have it timestamped [453].

The general protocol is given in [28]:

- (1) Alice signs a message.
- (2) Alice generates a header containing some identifying information. She concatenates the header with the signed message, signs that, and sends it to Trent.
- (3) Trent verifies the outside signature and confirms the identifying information. He adds a timestamp to Alice's signed message and the identifying information. Then he signs it all and sends it to both Alice and Bob.
- (4) Bob verifies Trent's signature, the identifying information, and Alice's signature.
- (5) Alice verifies the message Trent sent to Bob. If she did not originate the message, she speaks up quickly.

Another scheme uses Trent after the fact [209]. After receiving a signed message, Bob can send a copy to Trent for verification. Trent can attest to the validity of Alice's signature.

Applications of Digital Signatures

One of the earliest proposed applications of digital signatures was to facilitate the verification of nuclear test ban treaties [1454,1467]. The United States and the Soviet Union (anyone remember the Soviet Union?) permitted each other to put seismometers on the other's soil to monitor nuclear tests. The problem was that each country needed to assure itself that the host nation was not tampering with the data from the monitoring nation's seismometers. Simultaneously, the host nation needed to assure itself that the monitor was sending only the specific information needed for monitoring.

Conventional authentication techniques can solve the first problem, but only digital signatures can solve both problems. The host nation can read, but not alter, data from the seismometer, and the monitoring nation knows that the data has not been tampered with.

2.7 DIGITAL SIGNATURES WITH ENCRYPTION

By combining digital signatures with public-key cryptography, we develop a protocol that combines the security of encryption with the authenticity of digital signatures. Think of a letter from your mother: The signature provides proof of authorship and the envelope provides privacy.

- (1) Alice signs the message with her private key.

$$S_A(M)$$

- (2) Alice encrypts the signed message with Bob's public key and sends it to Bob.

$$E_B(S_A(M))$$

- (3) Bob decrypts the message with his private key.

$$D_B(E_B(S_A(M))) = S_A(M)$$

- (4) Bob verifies with Alice's public key and recovers the message.

$$V_A(S_A(M)) = M$$

Signing before encrypting seems natural. When Alice writes a letter, she signs it and then puts it in an envelope. If she put the letter in the envelope unsigned and then signed the envelope, then Bob might worry if the letter hadn't been covertly replaced. If Bob showed to Carol Alice's letter and envelope, Carol might accuse Bob of lying about which letter arrived in which envelope.

In electronic correspondence as well, signing before encrypting is a prudent practice [48]. Not only is it more secure—an adversary can't remove a signature from an encrypted message and add his own—but there are legal considerations: If the text

to be signed is not visible to the signer when he affixes his signature, then the signature may have little legal force [1312]. And there are some cryptanalytic attacks against this technique with RSA signatures (see Section 19.3).

There's no reason Alice has to use the same public-key/private-key key pair for encrypting and signing. She can have two key pairs: one for encryption and the other for signatures. Separation has its advantages: she can surrender her encryption key to the police without compromising her signature, one key can be escrowed (see Section 4.13) without affecting the other, and the keys can have different sizes and can expire at different times.

Of course, timestamps should be used with this protocol to prevent reuse of messages. Timestamps can also protect against other potential pitfalls, such as the one described below.

Resending the Message as a Receipt

Consider an implementation of this protocol, with the additional feature of confirmation messages. Whenever Bob receives a message, he returns it as a confirmation of receipt.

- (1) Alice signs a message with her private key, encrypts it with Bob's public key, and sends it to Bob.

$$E_B(S_A(M))$$

- (2) Bob decrypts the message with his private key and verifies the signature with Alice's public key, thereby verifying that Alice signed the message and recovering the message.

$$V_A(D_B(E_B(S_A(M)))) = M$$

- (3) Bob signs the message with his private key, encrypts it with Alice's public key, and sends it back to Alice.

$$E_A(S_B(M))$$

- (4) Alice decrypts the message with her private key and verifies the signature with Bob's public key. If the resultant message is the same one she sent to Bob, she knows that Bob received the message accurately.

If the same algorithm is used for both encryption and digital-signature verification there is a possible attack [506]. In these cases, the digital signature operation is the inverse of the encryption operation: $V_X = E_X$ and $S_X = D_X$.

Assume that Mallory is a legitimate system user with his own public and private key. Now, let's watch as he reads Bob's mail. First, he records Alice's message to Bob in step (1). Then, at some later time, he sends that message to Bob, claiming that it came from him (Mallory). Bob thinks that it is a legitimate message from Mallory, so he decrypts the message with his private key and then tries to verify Mallory's signature by decrypting it with Mallory's public key. The resultant message, which is pure gibberish, is:

$$E_M(D_B(E_B(D_A(M)))) = E_M(D_A(M))$$

Even so, Bob goes on with the protocol and sends Mallory a receipt:

$$E_M(D_B(E_M(D_A(M))))$$

Now, all Mallory has to do is decrypt the message with his private key, encrypt it with Bob's public key, decrypt it again with his private key, and encrypt it with Alice's public key. *Voilà!* Mallory has M .

It is not unreasonable to imagine that Bob may automatically send Mallory a receipt. This protocol may be embedded in his communications software, for example, and send receipts automatically. It is this willingness to acknowledge the receipt of gibberish that creates the insecurity. If Bob checked the message for comprehensibility before sending a receipt, he could avoid this security problem.

There are enhancements to this attack that allow Mallory to send Bob a different message from the one he eavesdropped on. Never sign arbitrary messages from other people or decrypt arbitrary messages and give the results to other people.

Foiling the Resend Attack

The attack just described works because the encrypting operation is the same as the signature-verifying operation and the decryption operation is the same as the signature operation. A secure protocol would use even a slightly different operation for encryption and digital signatures. Using different keys for each operation solves the problem, as does using different algorithms for each operation; as do time-stamps, which make the incoming message and the outgoing message different; as do digital signatures with one-way hash functions (see Section 2.6).

In general, then, the following protocol is secure as the public-key algorithm used:

- (1) Alice signs a message.
- (2) Alice encrypts the message and signature with Bob's public key (using a different encryption algorithm than for the signature) and sends it to Bob.
- (3) Bob decrypts the message with his private key.
- (4) Bob verifies Alice's signature.

Attacks against Public-Key Cryptography

In all these public-key cryptography protocols, I glossed over how Alice gets Bob's public key. Section 3.1 discusses this in detail, but it is worth mentioning here.

The easiest way to get someone's public key is from a secure database somewhere. The database has to be public, so that anyone can get anyone else's public key. The database also has to be protected from write-access by anyone except Trent; otherwise Mallory could substitute any public key for Bob's. After he did that, Bob couldn't read messages addressed to him, but Mallory could.

Even if the public keys are stored in a secure database, Mallory could still substitute one for another during transmission. To prevent this, Trent can sign each public key with his own private key. Trent, when used in this manner, is often known as a **Key Certification Authority** or **Key Distribution Center (KDC)**. In practical implementations, the KDC signs a compound message consisting of the user's

name, his public key, and any other important information about the user. This signed compound message is stored in the KDC's database. When Alice gets Bob's key, she verifies the KDC's signature to assure herself of the key's validity.

In the final analysis, this is not making things impossible for Mallory, only more difficult. Alice still has the KDC's public key stored somewhere. Mallory would have to substitute his own public key for that key, corrupt the database, and substitute his own keys for the valid keys (all signed with his private key as if he were the KDC), and then he's in business. But, even paper-based signatures can be forged if Mallory goes to enough trouble. Key exchange will be discussed in minute detail in Section 3.1.

2.8 RANDOM AND PSEUDO-RANDOM-SEQUENCE GENERATION

Why even bother with random-number generation in a book on cryptography? There's already a random-number generator built into most every compiler, a mere function call away. Why not use that? Unfortunately, those random-number generators are almost definitely not secure enough for cryptography, and probably not even very random. Most of them are embarrassingly bad.

Random-number generators are not random because they don't have to be. Most simple applications, like computer games, need so few random numbers that they hardly notice. However, cryptography is extremely sensitive to the properties of random-number generators. Use a poor random-number generator and you start getting weird correlations and strange results [1231,1238]. If you are depending on your random-number generator for security, weird correlations and strange results are the last things you want.

The problem is that a random-number generator doesn't produce a random sequence. It probably doesn't produce anything that looks even remotely like a random sequence. Of course, it is impossible to produce something truly random on a computer. Donald Knuth quotes John von Neumann as saying: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" [863]. Computers are deterministic beasts: Stuff goes in one end, completely predictable operations occur inside, and different stuff comes out the other end. Put the same stuff in on two separate occasions and the same stuff comes out both times. Put the same stuff into two identical computers, and the same stuff comes out of both of them. A computer can only be in a finite number of states (a large finite number, but a finite number nonetheless), and the stuff that comes out will always be a deterministic function of the stuff that went in and the computer's current state. That means that any random-number generator on a computer (at least, on a finite-state machine) is, by definition, periodic. Anything that is periodic is, by definition, predictable. And if something is predictable, it can't be random. A true random-number generator requires some random input; a computer can't provide that.

Pseudo-Random Sequences

The best a computer can produce is a **pseudo-random-sequence generator**. What's that? Many people have taken a stab at defining this formally, but I'll hand-wave here. A pseudo-random sequence is one that looks random. The sequence's period

should be long enough so that a finite sequence of reasonable length—that is, one that is actually used—is not periodic. If you need a billion random bits, don't choose a sequence generator that repeats after only sixteen thousand bits. These relatively short nonperiodic subsequences should be as indistinguishable as possible from random sequences. For example, they should have about the same number of ones and zeros, about half the runs (sequences of the same bit) should be of length one, one quarter of length two, one eighth of length three, and so on. They should not be compressible. The distribution of run lengths for zeros and ones should be the same [643,863,99,1357]. These properties can be empirically measured and then compared to statistical expectations using a chi-square test.

For our purposes, a sequence generator is pseudo-random if it has this property:

1. It looks random. This means that it passes all the statistical tests of randomness that we can find. (Start with the ones in [863].)

A lot of effort has gone into producing good pseudo-random sequences on computer. Discussions of generators abound in the academic literature, along with various tests of randomness. All of these generators are periodic (there's no escaping that); but with potential periods of 2^{256} bits and higher, they can be used for the largest applications.

The problem is still those weird correlations and strange results. Every pseudo-random-sequence generator is going to produce them if you use them in a certain way. And that's what a cryptanalyst will use to attack the system.

Cryptographically Secure Pseudo-Random Sequences

Cryptographic applications demand much more of a pseudo-random-sequence generator than do most other applications. Cryptographic randomness doesn't mean just statistical randomness, although that's part of it. For a sequence to be **cryptographically secure pseudo-random**, it must also have this property:

2. It is unpredictable. It must be computationally infeasible to predict what the next random bit will be, given complete knowledge of the algorithm or hardware generating the sequence and all of the previous bits in the stream.

Cryptographically secure pseudo-random sequences should not be compressible . . . unless you know the key. The key is generally the seed used to set the initial state of the generator.

Like any cryptographic algorithm, cryptographically secure pseudo-random-sequence generators are subject to attack. Just as it is possible to break an encryption algorithm, it is possible to break a cryptographically secure pseudo-random-sequence generator. Making generators resistant to attack is what cryptography is all about.

Real Random Sequences

Now we're drifting into the domain of philosophers. Is there such a thing as randomness? What is a random sequence? How do you know if a sequence is random? Is "101110100" more random than "101010101"? Quantum mechanics tells us that

there is honest-to-goodness randomness in the real world. But can we preserve that randomness in the deterministic world of computer chips and finite-state machines?

Philosophy aside, from our point of view a sequence generator is **real random** if it has this additional third property:

3. It cannot be reliably reproduced. If you run the sequence generator twice with the exact same input (at least as exact as humanly possible), you will get two completely unrelated random sequences.

The output of a generator satisfying these three properties will be good enough for a one-time pad, key generation, and any other cryptographic applications that require a truly random sequence generator. The difficulty is in determining whether a sequence is really random. If I repeatedly encrypt a string with DES and a given key, I will get a nice, random-looking output; you won't be able to tell that it's non-random unless you rent time on the NSA's DES cracker.

CHAPTER 3

Basic Protocols

3.1 KEY EXCHANGE

A common cryptographic technique is to encrypt each individual conversation with a separate key. This is called a session key, because it is used for only one particular communications session. As discussed in Section 8.5, session keys are useful because they only exist for the duration of the communication. How this common session key gets into the hands of the conversants can be a complicated matter.

Key Exchange with Symmetric Cryptography

This protocol assumes that Alice and Bob, users on a network, each share a secret key with the Key Distribution Center (KDC) [1260]—Trent in our protocols. These keys must be in place before the start of the protocol. (The protocol ignores the very real problem of how to distribute these secret keys; just assume they are in place and Mallory has no idea what they are.)

- (1) Alice calls Trent and requests a session key to communicate with Bob.
- (2) Trent generates a random session key. He encrypts two copies of it: one in Alice's key and the other in Bob's key. Trent sends both copies to Alice.
- (3) Alice decrypts her copy of the session key.
- (4) Alice sends Bob his copy of the session key.
- (5) Bob decrypts his copy of the session key.
- (6) Both Alice and Bob use this session key to communicate securely.

This protocol relies on the absolute security of Trent, who is more likely to be a trusted computer program than a trusted individual. If Mallory corrupts Trent, the whole network is compromised. He has all of the secret keys that Trent shares with

each of the users; he can read all past communications traffic that he has saved, and all future communications traffic. All he has to do is to tap the communications lines and listen to the encrypted message traffic.

The other problem with this system is that Trent is a potential bottleneck. He has to be involved in every key exchange. If Trent fails, that disrupts the entire system.

Key Exchange with Public-Key Cryptography

The basic hybrid cryptosystem was discussed in Section 2.5. Alice and Bob use public-key cryptography to agree on a session key, and use that session key to encrypt data. In some practical implementations, both Alice's and Bob's signed public keys will be available on a database. This makes the key-exchange protocol even easier, and Alice can send a secure message to Bob even if he has never heard of her:

- (1) Alice gets Bob's public key from the KDC.
- (2) Alice generates a random session key, encrypts it using Bob's public key, and sends it to Bob.
- (3) Bob then decrypts Alice's message using his private key.
- (4) Both of them encrypt their communications using the same session key.

Man-in-the-Middle Attack

While Eve cannot do better than try to break the public-key algorithm or attempt a ciphertext-only attack on the ciphertext, Mallory is a lot more powerful than Eve. Not only can he listen to messages between Alice and Bob, he can also modify messages, delete messages, and generate totally new ones. Mallory can imitate Bob when talking to Alice and imitate Alice when talking to Bob. Here's how the attack works:

- (1) Alice sends Bob her public key. Mallory intercepts this key and sends Bob his own public key.
- (2) Bob sends Alice his public key. Mallory intercepts this key and sends Alice his own public key.
- (3) When Alice sends a message to Bob, encrypted in "Bob's" public key, Mallory intercepts it. Since the message is really encrypted with his own public key, he decrypts it with his private key, re-encrypts it with Bob's public key, and sends it on to Bob.
- (4) When Bob sends a message to Alice, encrypted in "Alice's" public key, Mallory intercepts it. Since the message is really encrypted with his own public key, he decrypts it with his private key, re-encrypts it with Alice's public key, and sends it on to Alice.

Even if Alice's and Bob's public keys are stored on a database, this attack will work. Mallory can intercept Alice's database inquiry and substitute his own public

key for Bob's. He can do the same to Bob and substitute his own public key for Alice's. Or better yet, he can break into the database surreptitiously and substitute his key for both Alice's and Bob's. Then he simply waits for Alice and Bob to talk with each other, intercepts and modifies the messages, and he has succeeded.

This **man-in-the-middle attack** works because Alice and Bob have no way to verify that they are talking to each other. Assuming Mallory doesn't cause any noticeable network delays, the two of them have no idea that someone sitting between them is reading all of their supposedly secret communications.

Interlock Protocol

The **interlock protocol**, invented by Ron Rivest and Adi Shamir [1327], has a good chance of foiling the man-in-the-middle attack. Here's how it works:

- (1) Alice sends Bob her public key.
- (2) Bob sends Alice his public key.
- (3) Alice encrypts her message using Bob's public key. She sends half of the encrypted message to Bob.
- (4) Bob encrypts his message using Alice's public key. He sends half of the encrypted message to Alice.
- (5) Alice sends the other half of her encrypted message to Bob.
- (6) Bob puts the two halves of Alice's message together and decrypts it with his private key. Bob sends the other half of his encrypted message to Alice.
- (7) Alice puts the two halves of Bob's message together and decrypts it with her private key.

The important point is that half of the message is useless without the other half; it can't be decrypted. Bob cannot read any part of Alice's message until step (6); Alice cannot read any part of Bob's message until step (7). There are a number of ways to do this:

- If the encryption algorithm is a block algorithm, half of each block (e.g., every other bit) could be sent in each half message.
- Decryption of the message could be dependent on an initialization vector (see Section 9.3), which could be sent with the second half of the message.
- The first half of the message could be a one-way hash function of the encrypted message (see Section 2.4) and the encrypted message itself could be the second half.

To see how this causes a problem for Mallory, let's review his attempt to subvert the protocol. He can still substitute his own public keys for Alice's and Bob's in steps (1) and (2). But now, when he intercepts half of Alice's message in step (3), he

cannot decrypt it with his private key and re-encrypt it with Bob's public key. He must invent a totally new message and send half of it to Bob. When he intercepts half of Bob's message to Alice in step (4), he has the same problem. He cannot decrypt it with his private key and re-encrypt it with Alice's public key. He has to invent a totally new message and send half of it to Alice. By the time he intercepts the second halves of the real messages in steps (5) and (6), it is too late for him to change the new messages he invented. The conversation between Alice and Bob will necessarily be completely different.

Mallory could possibly get away with this scheme. If he knows Alice and Bob well enough to mimic both sides of a conversation between them, they might never realize that they are being duped. But surely this is much harder than sitting between the two of them, intercepting and reading their messages.

Key Exchange with Digital Signatures

Implementing digital signatures during a session-key exchange protocol circumvents this man-in-the-middle attack as well. Trent signs both Alice's and Bob's public keys. The signed keys include a signed certification of ownership. When Alice and Bob receive the keys, they each verify Trent's signature. Now they know that the public key belongs to that other person. The key exchange protocol can then proceed.

Mallory has serious problems. He cannot impersonate either Alice or Bob because he doesn't know either of their private keys. He cannot substitute his public key for either of theirs because, while he has one signed by Trent, it is signed as being Mallory's. All he can do is listen to the encrypted traffic go back and forth or disrupt the lines of communication and prevent Alice and Bob from talking.

This protocol uses Trent, but the risk of compromising the KDC is less than the first protocol. If Mallory compromises Trent (breaks into the KDC), all he gets is Trent's private key. This key enables him only to sign new keys; it does not let him decrypt any session keys or read any message traffic. To read the traffic, Mallory has to impersonate a user on the network and trick legitimate users into encrypting messages with his phony public key.

Mallory can launch that kind of attack. With Trent's private key, he can create phony signed keys to fool both Alice and Bob. Then, he can either exchange them in the database for real signed keys, or he can intercept users' database requests and reply with his phony keys. This enables him to launch a man-in-the-middle attack and read people's communications.

This attack will work, but remember that Mallory has to be able to intercept and modify messages. In some networks this is a lot more difficult than passively sitting on a network reading messages as they go by. On a broadcast channel, such as a radio network, it is almost impossible to replace one message with another—although the entire network can be jammed. On computer networks this is easier and seems to be getting easier every day. Consider IP spoofing, router attacks, and so forth; active attacks don't necessarily mean someone down a manhole with a datascope, and they are not limited to three-letter agencies.

Key and Message Transmission

Alice and Bob need not complete the key-exchange protocol before exchanging messages. In this protocol, Alice sends Bob the message, M , without any previous key exchange protocol:

- (1) Alice generates a random session key, K , and encrypts M using K .
 $E_K(M)$
 - (2) Alice gets Bob's public key from the database.
 - (3) Alice encrypts K with Bob's public key.
 $E_B(K)$
 - (4) Alice sends both the encrypted message and encrypted session key to Bob.
 $E_K(M), E_B(K)$
- For added security against man-in-the-middle attacks, Alice can sign the transmission.
- (5) Bob decrypts Alice's session key, K , using his private key.
 - (6) Bob decrypts Alice's message using the session key.

This hybrid system is how public-key cryptography is most often used in a communications system. It can be combined with digital signatures, timestamps, and any other security protocols.

Key and Message Broadcast

There is no reason Alice can't send the encrypted message to several people. In this example, Alice will send the encrypted message to Bob, Carol, and Dave:

- (1) Alice generates a random session key, K , and encrypts M using K .
 $E_K(M)$
- (2) Alice gets Bob's, Carol's, and Dave's public keys from the database.
- (3) Alice encrypts K with Bob's public key, encrypts K with Carol's public key, and then encrypts K with Dave's public key.
 $E_B(K), E_C(K), E_D(K)$
- (4) Alice broadcasts the encrypted message and all the encrypted keys to anybody who cares to receive it.
 $E_B(K), E_C(K), E_D(K), E_K(M)$
- (5) Only Bob, Carol, and Dave can decrypt the key, K , each using his or her private key.
- (6) Only Bob, Carol, and Dave can decrypt Alice's message using K .

This protocol can be implemented on a store-and-forward network. A central server can forward Alice's message to Bob, Carol, and Dave along with their partic-

ular encrypted key. The server doesn't have to be secure or trusted, since it will not be able to decrypt any of the messages.

3.2 AUTHENTICATION

When Alice logs into a host computer (or an automatic teller, or a telephone banking system, or any other type of terminal), how does the host know who she is? How does the host know she is not Eve trying to falsify Alice's identity? Traditionally, passwords solve this problem. Alice enters her password, and the host confirms that it is correct. Both Alice and the host know this secret piece of knowledge and the host requests it from Alice every time she tries to log in.

Authentication Using One-Way Functions

What Roger Needham and Mike Guy realized is that the host does not need to know the passwords; the host just has to be able to differentiate valid passwords from invalid passwords. This is easy with one-way functions [1599,526,1274,1121]. Instead of storing passwords, the host stores one-way functions of the passwords.

- (1) Alice sends the host her password.
- (2) The host performs a one-way function on the password.
- (3) The host compares the result of the one-way function to the value it previously stored.

Since the host no longer stores a table of everybody's valid password, the threat of someone breaking into the host and stealing the password list is mitigated. The list of passwords operated on by the one-way function is useless, because the one-way function cannot be reversed to recover the passwords.

Dictionary Attacks and Salt

A file of passwords encrypted with a one-way function is still vulnerable. In his spare time, Mallory compiles a list of the 1,000,000 most common passwords. He operates on all 1,000,000 of them with the one-way function and stores the results. If each password is about 8 bytes, the resulting file will be no more than 8 megabytes; it will fit on a few floppy disks. Now, Mallory steals an encrypted password file. He compares that file with his file of encrypted possible passwords and sees what matches.

This is a **dictionary attack**, and it's surprisingly successful (see Section 8.1). **Salt** is a way to make it more difficult. Salt is a random string that is concatenated with passwords before being operated on by the one-way function. Then, both the salt value and the result of the one-way function are stored in a database on the host. If the number of possible salt values is large enough, this practically eliminates a dictionary attack against commonly used passwords because Mallory has to generate the one-way hash for each possible salt value. This is a simple attempt at an initialization vector (see Section 9.3).

The point here is to make sure that Mallory has to do a trial encryption of each password in his dictionary every time he tries to break another person's password, rather than just doing one massive precomputation for all possible passwords.

A lot of salt is needed. Most UNIX systems use only 12 bits of salt. Even with that, Daniel Klein developed a password-guessing program that often cracks 40 percent of the passwords on a given host system within a week [847,848] (see Section 8.1). David Feldmeier and Philip Karn compiled a list of about 732,000 common passwords concatenated with each of 4096 possible salt values. They estimate that 30 percent of passwords on any given host can be broken with this list [561].

Salt isn't a panacea; increasing the number of salt bits won't solve everything. Salt only protects against general dictionary attacks on a password file, not against a concerted attack on a single password. It protects people who have the same password on multiple machines, but doesn't make poorly chosen passwords any better.

SKEY

SKEY is an authentication program that relies on a one-way function for its security. It's easy to explain.

To set up the system, Alice enters a random number, R . The computer computes $f(R)$, $f(f(R))$, $f(f(f(R)))$, and so on, about a hundred times. Call these numbers x_1 , x_2 , x_3 , ..., x_{100} . The computer prints out this list of numbers, and Alice puts it in her pocket for safekeeping. The computer also stores x_{101} , in the clear, in a login database next to Alice's name.

The first time Alice wants to log in, she types her name and x_{100} . The computer calculates $f(x_{100})$ and compares it with x_{101} ; if they match, Alice is authenticated. Then, the computer replaces x_{101} with x_{100} in the database. Alice crosses x_{100} off her list.

Every time Alice logs in, she enters the last uncrossed number on her list: x_i . The computer calculates $f(x_i)$ and compares it with x_{i+1} stored in its database. Eve can't get any useful information because each number is only used once, and the function is one-way. Similarly, the database is not useful to an attacker. Of course, when Alice runs out of numbers on her list, she has to reinitialize the system.

Authentication Using Public-Key Cryptography

Even with salt, the first protocol has serious security problems. When Alice sends her password to her host, anyone who has access to her data path can read it. She might be accessing her host through a convoluted transmission path that passes through four industrial competitors, three foreign countries, and two forward-thinking universities. Eve can be at any one of those points, listening to Alice's login sequence. If Eve has access to the processor memory of the host, she can see the password before the host hashes it.

Public-key cryptography can solve this problem. The host keeps a file of every user's public key; all users keep their own private keys. Here is a naïve attempt at a protocol. When logging in, the protocol proceeds as follows:

- (1) The host sends Alice a random string.
- (2) Alice encrypts the string with her private key and sends it back to the host, along with her name.
- (3) The host looks up Alice's public key in its database and decrypts the message using that public key.
- (4) If the decrypted string matches what the host sent Alice in the first place, the host allows Alice access to the system.

No one else has access to Alice's private key, so no one else can impersonate Alice. More important, Alice never sends her private key over the transmission line to the host. Eve, listening in on the interaction, cannot get any information that would enable her to deduce the private key and impersonate Alice.

The private key is both long and non-mnemonic, and will probably be processed automatically by the user's hardware or communications software. This requires an intelligent terminal that Alice trusts, but neither the host nor the communications path needs to be secure.

It is foolish to encrypt arbitrary strings—not only those sent by untrusted third parties, but under any circumstances at all. Attacks similar to the one discussed in Section 19.3 can be mounted. Secure proof-of-identity protocols take the following, more complicated, form:

- (1) Alice performs a computation based on some random numbers and her private key and sends the result to the host.
- (2) The host sends Alice a different random number.
- (3) Alice makes some computation based on the random numbers (both the ones she generated and the one she received from the host) and her private key, and sends the result to the host.
- (4) The host does some computation on the various numbers received from Alice and her public key to verify that she knows her private key.
- (5) If she does, her identity is verified.

If Alice does not trust the host any more than the host trusts Alice, then Alice will require the host to prove its identity in the same manner.

Step (1) might seem unnecessary and confusing, but it is required to prevent attacks against the protocol. Sections 21.1 and 21.2 mathematically describe several algorithms and protocols for proving identity. See also [935].

Mutual Authentication Using the Interlock Protocol

Alice and Bob are two users who want to authenticate each other. Each of them has a password that the other knows: Alice has P_A and Bob has P_B . Here's a protocol that will *not* work:

- (1) Alice and Bob trade public keys.
- (2) Alice encrypts P_A with Bob's public key and sends it to him.

- (3) Bob encrypts P_B with Alice's public key and sends it to her.
- (4) Alice decrypts what she received in step (2) and verifies that it is correct.
- (5) Bob decrypts what he received in step (3) and verifies that it is correct.

Mallory can launch a successful man-in-the-middle attack (see Section 3.1):

- (1) Alice and Bob trade public keys. Mallory intercepts both messages. He substitutes his public key for Bob's and sends it to Alice. Then he substitutes his public key for Alice's and sends it to Bob.
- (2) Alice encrypts P_A with "Bob's" public key and sends it to him. Mallory intercepts the message, decrypts P_A with his private key, re-encrypts it with Bob's public key and sends it on to him.
- (3) Bob encrypts P_B with "Alice's" public key and sends it to her. Mallory intercepts the message, decrypts P_B with his private key, re-encrypts it with Alice's public key, and sends it on to her.
- (4) Alice decrypts P_B and verifies that it is correct.
- (5) Bob decrypts P_A and verifies that it is correct.

Alice and Bob see nothing different. However, Mallory knows both P_A and P_B .

Donald Davies and William Price describe how the interlock protocol (described in Section 3.1) can defeat this attack [435]. Steve Bellovin and Michael Merritt discuss ways to attack this protocol [110]. If Alice is a user and Bob is a host, Mallory can pretend to be Bob, complete the beginning steps of the protocol with Alice, and then drop the connection. True artistry demands Mallory do this by simulating line noise or network failure, but the final result is that Mallory has Alice's password. He can then connect with Bob and complete the protocol, thus getting Bob's password, too.

The protocol can be modified so that Bob gives his password before Alice, under the assumption that the user's password is much more sensitive than the host's password. This falls to a more complicated attack, also described in [110].

SKID

SKID2 and SKID3 are symmetric cryptography identification protocols developed for RACE's RIPE project [1305] (See Section 25.7). They use a MAC (see Section 2.4) to provide security and both assume that both Alice and Bob share a secret key, K .

SKID2 allows Bob to prove his identity to Alice. Here's the protocol:

- (1) Alice chooses a random number, R_A . (The RIPE document specifies a 64-bit number). She sends it to Bob.
- (2) Bob chooses a random number, R_B . (The RIPE document specifies a 64-bit number). He sends Alice:

$$R_B, H_K(R_A, R_B, B)$$

H_K is the MAC. (The RIPE document suggests the RIPE-MAC function—see Section 18.14.) B is Bob's name.

- (3) Alice computes $H_K(R_A, R_B, B)$ and compares it with what she received from Bob. If the results are identical, then Alice knows that she is communicating with Bob.

SKID3 provides mutual authentication between Alice and Bob. Steps (1) through (3) are identical to SKID2, and then the protocol proceeds with:

- (4) Alice sends Bob:

$$H_K(R_B, A)$$

A is Alice's name.

- (5) Bob computes $H_K(R_B, A)$, and compares it with what he received from Alice. If the results are identical, then Bob knows that he is communicating with Alice.

This protocol is not secure against a man-in-the-middle attack. In general, a man-in-the-middle attack can defeat any protocol that doesn't involve a secret of some kind.

Message Authentication

When Bob receives a message from Alice, how does he know it is authentic? If Alice signed her message, this is easy. Alice's digital signature is enough to convince anyone that the message is authentic.

Symmetric cryptography provides some authentication. When Bob receives a message from Alice encrypted in their shared key, he knows it is from Alice. No one else knows their key. However, Bob has no way of convincing a third party of this fact. Bob can't show the message to Trent and convince him that it came from Alice. Trent can be convinced that the message came from either Alice or Bob (since no one else shared their secret key), but he has no way of knowing which one.

If the message is unencrypted, Alice could also use a MAC. This also convinces Bob that the message is authentic, but has the same problems as symmetric cryptography solutions.

3.3 AUTHENTICATION AND KEY EXCHANGE

These protocols combine authentication with key exchange to solve a general computer problem: Alice and Bob are on opposite ends of a network and want to talk securely. How can Alice and Bob exchange a secret key and at the same time each be sure that he or she is talking to the other and not to Mallory? Most of the protocols assume that Trent shares a different secret key with each participant, and that all of these keys are in place before the protocol begins.

The symbols used in these protocols are summarized in Table 3.1.

Wide-Mouth Frog

The Wide-Mouth Frog protocol [283,284] is probably the simplest symmetric key-management protocol that uses a trusted server. Both Alice and Bob share a secret

TABLE 3.1
Symbols used in authentication and key exchange protocols

A	Alice's name
B	Bob's name
E_A	Encryption with a key Trent shares with Alice
E_B	Encryption with a key Trent shares with Bob
I	Index number
K	A random session key
L	Lifetime
T_A, T_B	A timestamp
R_A, R_B	A random number, sometimes called a nonce , chosen by Alice and Bob respectively

key with Trent. The keys are just used for key distribution and not to encrypt any actual messages between users. Just by using two messages, Alice transfers a session key to Bob:

- (1) Alice concatenates a timestamp, Bob's name, and a random session key and encrypts the whole message with the key she shares with Trent. She sends this to Trent, along with her name.

$$A, E_A(T_A, B, K)$$

- (2) Trent decrypts the message from Alice. Then he concatenates a new timestamp, Alice's name, and the random session key; he encrypts the whole message with the key he shares with Bob. Trent sends to Bob:

$$E_B(T_B, A, K)$$

The biggest assumption made in this protocol is that Alice is competent enough to generate good session keys. Remember that random numbers aren't easy to generate; it might be more than Alice can be trusted to do properly.

Yahalom

In this protocol, both Alice and Bob share a secret key with Trent [283,284].

- (1) Alice concatenates her name and a random number, and sends it to Bob.

$$A, R_A$$

- (2) Bob concatenates Alice's name, Alice's random number, his own random number, and encrypts it with the key he shares with Trent. He sends this to Trent, along with his name.

$$B, E_B(A, R_A, R_B)$$

- (3) Trent generates two messages. The first consists of Bob's name, a random session key, Alice's random number, and Bob's random number, all encrypted with the key he shares with Alice. The second consists of

Alice's name and the random session key, encrypted with the key he shares with Bob. He sends both messages to Alice.

$$E_A\{B, K, R_A, R_B\}, E_B\{A, K\}$$

- (4) Alice decrypts the first message, extracts K , and confirms that R_A has the same value as it did in step (1). Alice sends Bob two messages. The first is the message received from Trent, encrypted with Bob's key. The second is R_B , encrypted with the session key.

$$E_B\{A, K\}, E_K\{R_B\}$$

- (5) Bob decrypts the message encrypted with his key, extracts K , and confirms that R_B has the same value as it did in step (2).

At the end, Alice and Bob are each convinced that they are talking to the other and not to a third party. The novelty here is that Bob is the first one to contact Trent, who only sends one message to Alice.

Needham-Schroeder

This protocol, invented by Roger Needham and Michael Schroeder [1159], also uses symmetric cryptography and Trent.

- (1) Alice sends a message to Trent consisting of her name, Bob's name, and a random number.

$$A, B, R_A$$

- (2) Trent generates a random session key. He encrypts a message consisting of a random session key and Alice's name with the secret key he shares with Bob. Then he encrypts Alice's random value, Bob's name, the key, and the encrypted message with the secret key he shares with Alice. Finally, he sends her the encrypted message:

$$E_A\{R_A, B, K, E_B\{K, A\}\}$$

- (3) Alice decrypts the message and extracts K . She confirms that R_A is the same value that she sent Trent in step (1). Then she sends Bob the message that Trent encrypted in his key.

$$E_B\{K, A\}$$

- (4) Bob decrypts the message and extracts K . He then generates another random value, R_B . He encrypts the message with K and sends it to Alice.

$$E_K\{R_B\}$$

- (5) Alice decrypts the message with K . She generates $R_B - 1$ and encrypts it with K . Then she sends the message back to Bob.

$$E_K\{R_B - 1\}$$

- (6) Bob decrypts the message with K and verifies that it is $R_B - 1$.

All of this fussing around with R_A and R_B and $R_B - 1$ is to prevent **replay attacks**. In this attack, Mallory can record old messages and then use them later in an attempt to subvert the protocol. The presence of R_A in step (2) assures Alice that

Trent's message is legitimate and not a replay of a response from a previous execution of the protocol. When Alice successfully decrypts R_B and sends Bob $R_B - 1$ in step (5), Bob is ensured that Alice's messages are not replays from an earlier execution of the protocol.

The major security hole in this protocol is that old session keys are valuable. If Mallory gets access to an old K , he can launch a successful attack [461]. All he has to do is record Alice's messages to Bob in step (3). Then, once he has K , he can pretend to be Alice:

- (1) Mallory sends Bob the following message:

$$E_B(K, A)$$

- (2) Bob extracts K , generates R_B , and sends Alice:

$$E_K(R_B)$$

- (3) Mallory intercepts the message, decrypts it with K , and sends Bob:

$$E_K(R_B - 1)$$

- (4) Bob verifies that "Alice's" message is $R_B - 1$.

Now, Mallory has Bob convinced that he is Alice.

A stronger protocol, using timestamps, can defeat this attack [461, 456]. A timestamp is added to Trent's message in step (2) encrypted with Bob's key: $E_B(K, A, T)$. Timestamps require a secure and accurate system clock—not a trivial problem in itself.

If the key Trent shares with Alice is ever compromised, the consequences are drastic. Mallory can use it to obtain session keys to talk with Bob (or anyone else he wishes to talk to). Even worse, Mallory can continue to do this even after Alice changes her key [90].

Needham and Schroeder attempted to correct these problems in a modified version of their protocol [1160]. Their new protocol is essentially the same as the Otway-Rees protocol, published in the same issue of the same journal.

Otway-Rees

This protocol also uses symmetric cryptography [1224].

- (1) Alice generates a message consisting of an index number, her name, Bob's name, and a random number, all encrypted in the key she shares with Trent. She sends this message to Bob along with the index number, her name, and his name:

$$I, A, B, E_A(R_A, I, A, B)$$

- (2) Bob generates a message consisting of a new random number, the index number, Alice's name, and Bob's name, all encrypted in the key he shares with Trent. He sends it to Trent, along with Alice's encrypted message, the index number, her name, and his name:

$$I, A, B, E_A(R_A, I, A, B), E_B(R_B, I, A, B)$$

- (3) Trent generates a random session key. Then he creates two messages. One is Alice's random number and the session key, encrypted in the key he shares with Alice. The other is Bob's random number and the session key, encrypted in the key he shares with Bob. He sends these two messages, along with the index number, to Bob:

$$I, E_A(R_A, K), E_B(R_B, K)$$

- (4) Bob sends Alice the message encrypted in her key, along with the index number:

$$I, E_A(R_A, K)$$

- (5) Alice decrypts the message to recover her key and random number. She then confirms that both have not changed in the protocol.

Assuming that all the random numbers match, and the index number hasn't changed along the way, Alice and Bob are now convinced of each other's identity, and they have a secret key with which to communicate.

Kerberos

Kerberos is a variant of Needham-Schroeder and is discussed in detail in Section 24.5. In the basic Kerberos Version 5 protocol, Alice and Bob each share keys with Trent. Alice wants to generate a session key for a conversation with Bob.

- (1) Alice sends a message to Trent with her identity and Bob's identity.

$$A, B$$

- (2) Trent generates a message with a timestamp, a lifetime, L , a random session key, and Alice's identity. He encrypts this in the key he shares with Bob. Then he takes the timestamp, the lifetime, the session key, and Bob's identity, and encrypts these in the key he shares with Alice. He sends both encrypted messages to Alice.

$$E_A(T, L, K, B), E_B(T, L, K, A)$$

- (3) Alice generates a message with her identity and the timestamp, encrypts it in K , and sends it to Bob. Alice also sends Bob the message encrypted in Bob's key from Trent.

$$E_K(A, T), E_B(T, L, K, A)$$

- (4) Bob sends a message consisting of the timestamp plus one, encrypts it in K , and sends it to Alice.

$$E_K(T + 1)$$

This protocol works, but it assumes that everyone's clocks are synchronized with Trent's clock. In practice, the effect is obtained by synchronizing clocks to within a few minutes of a secure time server and detecting replays within the time interval.

Neuman-Stubblebine

Whether by system faults or by sabotage, clocks can become unsynchronized. If the clocks get out of sync, there is a possible attack against most of these protocols

[644]. If the sender's clock is ahead of the receiver's clock, Mallory can intercept a message from the sender and replay it later when the timestamp becomes current at the receiver's site. This attack is called **suppress-replay** and can have irritating consequences.

This protocol, first presented in [820] and corrected in [1162] attempts to counter the suppress-replay attack. It is an enhancement to Yahalom and is an excellent protocol.

- (1) Alice concatenates her name and a random number and sends it to Bob.

$$A, R_A$$

- (2) Bob concatenates Alice's name, her random number, and a timestamp, and encrypts with the key he shares with Trent. He sends it to Trent along with his name and a new random number.

$$B, R_B, E_B(A, R_A, T_B)$$

- (3) Trent generates a random session key. Then he creates two messages. The first is Bob's name, Alice's random number, a random session key, and the timestamp, all encrypted with the key he shares with Alice. The second is Alice's name, the session key, and the timestamp, all encrypted with the key he shares with Bob. He sends these both to Alice, along with Bob's random number.

$$E_A(B, R_A, K, T_B), E_A(B, K, T_B), R_B$$

- (4) Alice decrypts the message encrypted with her key, extracts K , and confirms that R_A has the same value as it did in step (1). Alice sends Bob two messages. The first is the message received from Trent, encrypted with Bob's key. The second is R_B , encrypted with the session key.

$$E_B(A, K, T_B), E_K(R_B)$$

- (5) Bob decrypts the message encrypted with his key, extracts K , and confirms that T_B and R_B have the same value they did in step (2).

Assuming both random numbers and the timestamp match, Alice and Bob are convinced of one another's identity and share a secret key. Synchronized clocks are not required because the timestamp is only relative to Bob's clock; Bob only checks the timestamp he generated himself.

One nice thing about this protocol is that Alice can use the message she received from Trent for subsequent authentication with Bob, within some predetermined time limit. Assume that Alice and Bob completed the above protocol, communicated, and then terminated the connection. Alice and Bob can reauthenticate in three steps, without having to rely on Trent.

- (1) Alice sends Bob the message Trent sent her in step (3) and a new random number.

$$E_B(A, K, T_B), R'_A$$

- (2) Bob sends Alice another new random number, and Alice's new random number encrypted in their session key.

$$R'_B E_K(R'_A)$$

- (3) Alice sends Bob his new random number, encrypted in their session key.

$$E_K(R'_B)$$

The new random numbers prevent replay attacks.

DASS

The Distributed Authentication Security Service (DASS) protocols, developed at Digital Equipment Corporation, also provide for mutual authentication and key exchange [604,1519,1518]. Unlike the previous protocols, DASS uses both public-key and symmetric cryptography. Alice and Bob each have a private key. Trent has signed copies of their public keys.

- (1) Alice sends a message to Trent, consisting of Bob's name.

$$B$$

- (2) Trent sends Alice Bob's public key, K_B , signed with Trent's private key, T . The signed message includes Bob's name.

$$S_T(B, K_B)$$

- (3) Alice verifies Trent's signature to confirm that the key she received is actually Bob's public key. She generates a random session key, and a random public-key/private-key key pair: K_p . She encrypts a timestamp with K . Then she signs a key lifetime, L , her name, and K_p with her private key, K_A . Finally, she encrypts K with Bob's public key, and signs it with K_p . She sends all of this to Bob.

$$E_K(T_A), S_{K_A}(L, A, K_p), S_{K_p}(E_{K_B}(K))$$

- (4) Bob sends a message to Trent (this may be a different Trent), consisting of Alice's name.

$$A$$

- (5) Trent sends Bob Alice's public key, signed in Trent's private key. The signed message includes Alice's name.

$$S_T(A, K_A)$$

- (6) Bob verifies Trent's signature to confirm that the key he received is actually Alice's public key. He then verifies Alice's signature and recovers K_p . He verifies the signature and uses his private key to recover K . Then he decrypts T_A to make sure this is a current message.

- (7) If mutual authentication is required, Bob encrypts a new timestamp with K , and sends it to Alice.

$$E_K(T_B)$$

- (8) Alice decrypts T_B with K to make sure that the message is current.

SPX, a product by DEC, is based on DASS. Additional information can be found in [34].

Denning-Sacco

This protocol also uses public-key cryptography [461]. Trent keeps a database of everyone's public keys.

- (1) Alice sends a message to Trent with her identity and Bob's identity:

$$A, B$$

- (2) Trent sends Alice Bob's public key, K_B , signed with Trent's private key, T . Trent also sends Alice her own public key, K_A , signed with his private key.

$$S_T(B, K_B), S_T(A, K_A)$$

- (3) Alice sends Bob a random session key and a timestamp, signed in her private key and encrypted in Bob's public key, along with both signed public keys.

$$E_B(S_A(K, T_A)), S_T(B, K_B), S_T(A, K_A)$$

- (4) Bob decrypts Alice's message with his private key and then verifies Alice's signature with her public key. He checks to make sure that the timestamp is still valid.

At this point both Alice and Bob have K , and can communicate securely.

This looks good, but it isn't. After completing the protocol with Alice, Bob can then masquerade as Alice [5]. Watch:

- (1) Bob sends his name and Carol's name to Trent

$$B, C$$

- (2) Trent sends Bob both Bob's and Carol's signed public keys.

$$S_T(B, K_B), S_T(C, K_C)$$

- (3) Bob sends Carol the signed session key and timestamp he previously received from Alice, encrypted with Carol's public key, along with Alice's certificate and Carol's certificate.

$$E_C(S_A(K, T_A)), S_T(A, K_A), S_T(C, K_C)$$

- (4) Carol decrypts Alice's message with her private key and then verifies Alice's signature with her public key. She checks to make sure that the timestamp is still valid.

Carol now thinks she is talking to Alice; Bob has successfully fooled her. In fact, Bob can fool everyone on the network until the timestamp expires.

This is easy to fix. Add the names inside the encrypted message in step (3):

$$E_B(S_A(A, B, K, T_A)), S_T(A, K_A), S_T(B, K_B)$$

Now Bob can't replay the old message to Carol, because it is clearly meant for communication between Alice and Bob.

Woo-Lam

This protocol also uses public-key cryptography [1610, 1611]:

- (1) Alice sends a message to Trent with her identity and Bob's identity:

$$A, B$$

- (2) Trent sends Alice Bob's public key, K_B , signed with Trent's private key, T :

$$S_T(K_B)$$

- (3) Alice verifies Trent's signature. Then she sends Bob her name and a random number, encrypted with Bob's public key.

$$E_{K_B}(A, R_A)$$

- (4) Bob sends Trent his name, Alice's name, and Alice's random number encrypted with Trent's public key, K_T .

$$A, B, E_{K_T}(R_A)$$

- (5) Trent sends Bob Alice's public key, K_A , signed with Trent's private key. He also sends him Alice's random number, a random session key, Alice's name, and Bob's name, all signed with Trent's private key and encrypted with Bob's public key.

$$S_T(K_A), E_{K_B}(S_T(R_A, K, A, B))$$

- (6) Bob verifies Trent's signatures. Then he sends Alice the second part of Trent's message from step (5) and a new random number—all encrypted in Alice's public key.

$$E_{K_A}(S_T(R_A, K, A, B), R_B)$$

- (7) Alice verifies Trent's signature and her random number. Then she sends Bob the second random number, encrypted in the session key.

$$E_K(R_B)$$

- (8) Bob decrypts his random number and verifies that it unchanged.

Other Protocols

There are many other protocols in the literature. The X.509 protocols are discussed in Section 24.9, KryptoKnight is discussed in Section 24.6, and Encrypted Key Exchange is discussed in Section 22.5.

Another new public-key protocol is Kuperee [694]. And work is being done on protocols that use **beacons**, a trusted node on a network that continuously broadcasts authenticated nonces [783].

Lessons Learned

There are some important lessons in the previous protocols, both those which have been broken and those which have not:

- Many protocols failed because the designers tried to be too clever. They optimized their protocols by leaving out important pieces: names, random numbers, and so on. The remedy is to make everything explicit [43,44].
- Trying to optimize is an absolute tar pit and depends a whole lot on the assumptions you make. For example: If you have authenticated time, you can do a whole lot of things you can't do if you don't.

- The protocol of choice depends on the underlying communications architecture. Do you want to minimize the size of messages or the number of messages? Can all parties talk with each other or can only a few of them?

It's questions like these that led to the development of formal methods for analyzing protocols.

3.4 FORMAL ANALYSIS OF AUTHENTICATION AND KEY-EXCHANGE PROTOCOLS

The problem of establishing secure session keys between pairs of computers (and people) on a network is so fundamental that it has led to a great deal of research. Some of the research focused on the development of protocols like the ones discussed in Sections 3.1, 3.2, and 3.3. This, in turn, has led to a greater and more interesting problem: the formal analysis of authentication and key-exchange protocols. People have found flaws in seemingly secure protocols years after they were proposed, and researchers wanted tools that could prove a protocol's security from the start. Although much of this work can apply to general cryptographic protocols, the emphasis in research is almost exclusively on authentication and key exchange.

There are four basic approaches to the analysis of cryptographic protocols [1045]:

1. Model and verify the protocol using specification languages and verification tools not specifically designed for the analysis of cryptographic protocols.
2. Develop expert systems that a protocol designer can use to develop and investigate different scenarios.
3. Model the requirements of a protocol family using logics for the analysis of knowledge and belief.
4. Develop a formal method based on the algebraic term-rewriting properties of cryptographic systems.

A full discussion on these four approaches and the research surrounding them is well beyond the scope of this book. See [1047,1355] for a good introduction to the topic; I am only going to touch on the major contributions to the field.

The first approach treats a cryptographic protocol as any other computer program and attempts to prove correctness. Some researchers represent a protocol as a finite-state machine [1449,1565], others use extensions of first-order predicate calculus [822], and still others use specification languages to analyze protocols [1566]. However, proving correctness is not the same as proving security and this approach fails to detect many flawed protocols. Although it was widely studied at first, most of the work in this area has been redirected as the third approach gained popularity.

The second approach uses expert systems to determine if a protocol can reach an undesirable state (the leaking of a key, for example). While this approach better identifies flaws, it neither guarantees security nor provides techniques for develop-

ing attacks. It is good at determining whether a protocol contains a given flaw, but is unlikely to discover unknown flaws in a protocol. Examples of this approach can be found in [987,1521]; [1092] discusses a rule-based system developed by the U.S. military, called the Interrogator.

The third approach is by far the most popular, and was pioneered by Michael Burrows, Martin Abadi, and Roger Needham. They developed a formal logic model for the analysis of knowledge and belief, called **BAN logic** [283,284]. BAN logic is the most widely used logic for analyzing authentication protocols. It assumes that authentication is a function of integrity and freshness, and uses logical rules to trace both of those attributes through the protocol. Although many variants and extensions have been proposed, most protocol designers still refer back to the original work.

BAN logic doesn't provide a proof of security; it can only reason about authentication. It has a simple, straightforward logic that is easy to apply and still useful for detecting flaws. Some of the statements in BAN logic include:

Alice believes X . (Alice acts as though X is true.)

Alice sees X . (Someone has sent a message containing X to Alice, who can read and repeat X —possibly after decrypting it.)

Alice said X . (At some time, Alice sent a message that includes the statement X . It is not known how long ago the message was sent or even that it was sent during the current run of the protocol. It is known that Alice believed X when she said it.)

X is fresh. (X has not been sent in a message at any time before the current run of the protocol.)

And so on. BAN logic also provides rules for reasoning about belief in a protocol. These rules can then be applied to the logical statements about the protocol to prove things or answer questions about the protocol. For example, one rule is the message-meaning rule:

IF Alice believes that Alice and Bob share a secret key, K , and Alice sees X , encrypted under K , and Alice did not encrypt X under K , THEN Alice believes that Bob once said X .

Another rule is the nonce-verification rule:

IF Alice believes that X could have been uttered only recently and that Bob once said X , THEN Alice believes that Bob believes X .

There are four steps in BAN analysis:

- (1) Convert the protocol into idealized form, using the statements previously described.
- (2) Add all assumptions about the initial state of the protocol.

- (3) Attach logical formulas to the statements: assertions about the state of the system after each statement.
- (4) Apply the logical postulates to the assertions and assumptions to discover the beliefs held by the parties in the protocol.

The authors of BAN logic “view the idealized protocols as clearer and more complete specifications than traditional descriptions found in the literature. . . .” [283,284]. Others are not so impressed and criticize this step because it may not accurately reflect the real protocol [1161,1612]. Further debate is in [221,1557]. Other critics try to show that BAN logic can deduce characteristics about protocols that are obviously false [1161]—see [285,1509] for a rebuttal—and that BAN logic deals only with trust and not security [1509]. More debate is in [1488, 706,1002].

Despite these criticisms, BAN logic has been a success. It has found flaws in several protocols, including Needham-Schroeder and an early draft of a CCITT X.509 protocol [303]. It has uncovered redundancies in many protocols, including Yahalom, Needham-Schroeder, and Kerberos. Many published papers use BAN logic to make claims about their protocol’s security [40,1162,73].

Other logic systems have been published, some designed as extensions to BAN logic [645,586,1556,828] and others based on BAN to correct perceived weaknesses [1488,1002]. The most successful of these is GNY [645], although it has some shortcomings [40]. Probabilistic beliefs were added to BAN logic, with mixed success, by [292,474]. Other formal logics are [156,798,288]; [1514] attempts to combine the features of several logics. And [1124,1511] present logics where beliefs can change over time.

The fourth approach to the analysis of cryptographic protocols models the protocol as an algebraic system, expresses the state of the participants’ knowledge about the protocol, and then analyzes the attainability of certain states. This approach has not received as much attention as formal logics, but that is changing. It was first used by Michael Merritt [1076], who showed that an algebraic model can be used to analyze cryptographic protocols. Other approaches are in [473,1508,1530,1531,1532, 1510,1612].

The Navy Research Laboratory’s (NRL) Protocol Analyzer is probably the most successful application of these techniques [1512,823,1046,1513]; it has been used to discover both new and known flaws in a variety of protocols [1044,1045,1047]. The Protocol Analyzer defines the following actions:

- Accept (Bob, Alice, M , N). (Bob accepts the message M as from Alice during Bob’s local round N .)
- Learn (Eve, M). (Eve learns M .)
- Send (Alice, Bob, Q , M). (Alice sends M to Bob in response to query, Q .)
- Request (Bob, Alice, Q , N). (Bob sends Q to Alice during Bob’s local round N .)

From these actions, requirements can be specified. For example:

- If Bob accepted message M from Alice at some point in the past, then Eve did not learn M at some point in the past.
- If Bob accepted message M from Alice in Bob's local round N , then Alice sent M to Bob as a response to a query in Bob's local round N .

To use the NRL Protocol Analyzer, a protocol must be specified using the previous constructs. Then, there are four phases of analysis: defining transition rules for honest participants, describing operations available to all—honest and dishonest—participants, describing the basic building blocks of the protocol, and describing the reduction rules. The point of all this is to show that a given protocol meets its requirements. Tools like the NRL Protocol Analyzer could eventually lead to a protocol that can be proven secure.

While much of the work in formal methods involves applying the methods to existing protocols, there is some push towards using formal methods to design the protocols in the first place. Some preliminary steps in this direction are [711]. The NRL Protocol Analyzer also attempts to do this [1512,222,1513].

The application of formal methods to cryptographic protocols is still a fairly new idea and it's really hard to figure out where it is headed. At this point, the weakest link seems to be the formalization process.

3.5 MULTIPLE-KEY PUBLIC-KEY CRYPTOGRAPHY

Public-key cryptography uses two keys. A message encrypted with one key can be decrypted with the other. Usually one key is private and the other is public. However, let's assume that Alice has one key and Bob has the other. Now Alice can encrypt a message so that only Bob can decrypt it, and Bob can encrypt a message so that only Alice can read it.

This concept was generalized by Colin Boyd [217]. Imagine a variant of public-key cryptography with three keys: K_A , K_B , and K_C , distributed as shown in Table 3.2.

Alice can encrypt a message with K_A so that Ellen, with K_B and K_C , can decrypt it. So can Bob and Carol in collusion. Bob can encrypt a message so that Frank can read

TABLE 3.2
Three-Key Key Distribution

Alice	K_A
Bob	K_B
Carol	K_C
Dave	K_A and K_B
Ellen	K_B and K_C
Frank	K_C and K_A

it, and Carol can encrypt a message so that Dave can read it. Dave can encrypt a message with K_A so that Ellen can read it, with K_B so that Frank can read it, or with both K_A and K_B so that Carol can read it. Similarly, Ellen can encrypt a message so that either Alice, Dave, or Frank can read it. All the possible combinations are summarized in Table 3.3; there are no other ones.

This can be extended to n keys. If a given subset of the keys is used to encrypt the message, then the other keys are required to decrypt the message.

Broadcasting a Message

Imagine that you have 100 operatives out in the field. You want to be able to send messages to subsets of them, but don't know which subsets in advance. You can either encrypt the message separately for each person or give out keys for every possible combination of people. The first option requires a lot of messages; the second requires a lot of keys.

Multiple-key cryptography is much easier. We'll use three operatives: Alice, Bob, and Carol. You give Alice K_A and K_B , Bob K_B and K_C , and Carol K_C and K_A . Now you can talk to any subset you want. If you want to send a message so that only Alice can read it, encrypt it with K_C . When Alice receives the message, she decrypts it with K_A and then K_B . If you want to send a message so that only Bob can read it, encrypt it with K_A so that only Carol can read it, with K_B . If you want to send a message so that both Alice and Bob can read it, encrypt it with K_A and K_C , and so on.

This might not seem exciting, but with 100 operatives it is quite efficient. Individual messages mean a shared key with each operative (100 keys total) and each message. Keys for every possible subset means $2^{100} - 2$ different keys (messages to all operatives and messages to no operatives are excluded). This scheme needs only one encrypted message and 100 different keys. The drawback of this scheme is that you also have to broadcast which subset of operatives can read the message, otherwise each operative would have to try every combination of possible keys looking for the correct one. Even just the names of the intended recipients may be significant. At least for the straightforward implementation of this, everyone gets a really large amount of key data.

There are other techniques for message broadcasting, some of which avoid the previous problem. These are discussed in Section 22.7.

TABLE 3.3
Three-Key Message Encryption

Encrypted with Keys:	Must be Decrypted with Keys:
K_A	K_B and K_C
K_B	K_A and K_C
K_C	K_A and K_B
K_A and K_B	K_C
K_A and K_C	K_B
K_B and K_C	K_A

3.6 SECRET SPLITTING

Imagine that you've invented a new, extra gooey, extra sweet, cream filling or a burger sauce that is even more tasteless than your competitors'. This is important, you have to keep it secret. You could tell only your most trusted employees the exact mixture of ingredients, but what if one of them defects to the competition? There goes the secret, and before long every grease palace on the block will be making burgers with sauce as tasteless as yours.

This calls for **secret splitting**. There are ways to take a message and divide it up into pieces [551]. Each piece by itself means nothing, but put them together and the message appears. If the message is the recipe and each employee has a piece, then only together can they make the sauce. If any employee resigns with his single piece of the recipe, his information is useless by itself.

The simplest sharing scheme splits a message between two people. Here's a protocol in which Trent can split a message between Alice and Bob:

- (1) Trent generates a random-bit string, R , the same length as the message, M .
- (2) Trent XORs M with R to generate S .

$$M \oplus R = S$$

- (3) Trent gives R to Alice and S to Bob.

To reconstruct the message, Alice and Bob have only one step to do:

- (4) Alice and Bob XOR their pieces together to reconstruct the message:

$$R \oplus S = M$$

This technique, if done properly, is absolutely secure. Each piece, by itself, is absolutely worthless. Essentially, Trent is encrypting the message with a one-time pad and giving the ciphertext to one person and the pad to the other person. Section 1.5 discusses one-time pads; they have perfect security. No amount of computing power can determine the message from one of the pieces.

It is easy to extend this scheme to more people. To split a message among more than two people, XOR more random-bit strings into the mixture. In this example, Trent divides up a message into four pieces:

- (1) Trent generates three random-bit strings, R , S , and T , the same length as the message, M .
- (2) Trent XORs M with the three strings to generate U :

$$M \oplus R \oplus S \oplus T = U$$

- (3) Trent gives R to Alice, S to Bob, T to Carol, and U to Dave.

Alice, Bob, Carol, and Dave, working together, can reconstruct the message:

- (4) Alice, Bob, Carol, and Dave get together and compute:

$$R \oplus S \oplus T \oplus U = M$$

This is an adjudicated protocol. Trent has absolute power and can do whatever he wants. He can hand out gibberish and claim that it is a valid piece of the secret; no one will know it until they try to reconstruct the secret. He can hand out a piece to Alice, Bob, Carol, and Dave, and later tell everyone that only Alice, Carol, and Dave are needed to reconstruct the secret, and then fire Bob. But since this is Trent's secret to divide up, this isn't a problem.

However, this protocol has a problem: If any of the pieces gets lost and Trent isn't around, so does the message. If Carol, who has a piece of the sauce recipe, goes to work for the competition and takes her piece with her, the rest of them are out of luck. She can't reproduce the recipe, but neither can Alice, Bob, and Dave working together. Her piece is as critical to the message as every other piece combined. All Alice, Bob, or Dave know is the length of the message—nothing more. This is true because R , S , T , U , and M all have the same length; seeing anyone of them gives the length of M . Remember, M isn't being split in the normal sense of the word; it is being XORed with random values.

3.7 SECRET SHARING

You're setting up a launch program for a nuclear missile. You want to make sure that no single raving lunatic can initiate a launch. You want to make sure that no two raving lunatics can initiate a launch. You want at least three out of five officers to be raving lunatics before you allow a launch.

This is easy to solve. Make a mechanical launch controller. Give each of the five officers a key and require that at least three officers stick their keys in the proper slots before you'll allow them to blow up whomever we're blowing up this week. (If you're really worried, make the slots far apart and require the officers to insert the keys simultaneously—you wouldn't want an officer who steals two keys to be able to vaporize Toledo.)

We can get even more complicated. Maybe the general and two colonels are authorized to launch the missile, but if the general is busy playing golf then five colonels are required to initiate a launch. Make the launch controller so that it requires five keys. Give the general three keys and the colonels one each. The general together with any two colonels can launch the missile; so can the five colonels. However, a general and one colonel cannot; neither can four colonels.

A more complicated sharing scheme, called a **threshold scheme**, can do all of this and more—mathematically. At its simplest level, you can take any message (a secret recipe, launch codes, your laundry list, etc.) and divide it into n pieces, called **shadows** or shares, such that any m of them can be used to reconstruct the message. More precisely, this is called an **(m,n) -threshold scheme**.

With a $(3,4)$ -threshold scheme, Trent can divide his secret sauce recipe among Alice, Bob, Carol, and Dave, such that any three of them can put their shadows together and reconstruct the message. If Carol is on vacation, Alice, Bob, and Dave can do it. If Bob gets run over by a bus, Alice, Carol, and Dave can do it. However, if Bob gets run over by a bus while Carol is on vacation, Alice and Dave can't reconstruct the message by themselves.

General threshold schemes are even more versatile. Any sharing scenario you can imagine can be modeled. You can divide a message among the people in your building so that to reconstruct it, you need seven people from the first floor and five people from the second floor, unless there is someone from the third floor involved, in which case you only need that person and three people from the first floor and two people from the second floor, unless there is someone from the fourth floor involved, in which case you need that person and one person from the third floor, or that person and two people from the first floor and one person from the second floor, unless there is . . . well, you get the idea.

This idea was invented independently by Adi Shamir [1414] and George Blakley [182] and studied extensively by Gus Simmons [1466]. Several different algorithms are discussed in Section 23.2.

Secret Sharing with Cheaters

There are many ways to cheat with a threshold scheme. Here are just a few of them.

Scenario 1: Colonels Alice, Bob, and Carol are in a bunker deep below some isolated field. One day, they get a coded message from the president: "Launch the missiles. We're going to eradicate the last vestiges of neural network research in the country." Alice, Bob, and Carol reveal their shadows, but Carol enters a random number. She's actually a pacifist and doesn't want the missiles launched. Since Carol doesn't enter the correct shadow, the secret they recover is the wrong secret. The missiles stay in their silos. Even worse, no one knows why. Alice and Bob, even if they work together, cannot prove that Carol's shadow is invalid.

Scenario 2: Colonels Alice and Bob are sitting in the bunker with Mallory. Mallory has disguised himself as a colonel and none of the others is the wiser. The same message comes in from the president, and everyone reveals their shadows. "Bwa-ha-ha!" shouts Mallory. "I faked that message from the president. Now I know both of your shadows." He races up the staircase and escapes before anyone can catch him.

Scenario 3: Colonels Alice, Bob, and Carol are sitting in the bunker with Mallory, who is again disguised. (Remember, Mallory doesn't have a valid shadow.) The same message comes in from the president and everyone reveals their shadows. Mallory reveals his shadow only after he has heard the other three. Since only three shadows are needed to reconstruct the secret, he can quickly create a valid shadow and reveals that. Now, not only does he know the secret, but no one realizes that he isn't part of the scheme.

Some protocols that handle these sorts of cheaters are discussed in Section 23.2.

Secret Sharing without Trent

A bank wants its vault to open only if three out of five officers enter their keys. This sounds like a basic (3,5)-threshold scheme, but there's a catch. No one is to know the entire secret. There is no Trent to divide the secret up into five pieces. There are protocols by which the five officers can create a secret and each get a piece, such that none of the officers knows the secret until they all reconstruct it. I'm not going to discuss these protocols in this book; see [756] for details.

Sharing a Secret without Revealing the Shares

These schemes have a problem. When everyone gets together to reconstruct their secret, they reveal their shares. This need not be the case. If the shared secret is a private key (to a digital signature, for example), then n shareholders can each complete a partial signature of the document. After the n th partial signature, the document has been signed with the shared private key and none of the shareholders learns any other shares. The point is that the secret can be reused, and you don't need a trusted processor to handle it. This concept is explored further by Yvo Desmedt and Yair Frankel [483,484].

Verifiable Secret Sharing

Trent gives Alice, Bob, Carol, and Dave each a share or at least he says he does. The only way any of them know if they have a valid share is to try to reconstruct the secret. Maybe Trent sent Bob a bogus share or Bob accidentally received a bad share through communications error. Verifiable secret sharing allows each of them to individually verify that they have a valid share, without having to reconstruct the secret [558,1235].

Secret-Sharing Schemes with Prevention

A secret is divided up among 50 people so that any 10 can get together and reconstruct the secret. That's easy. But, can we implement the same secret-sharing scheme with the added constraint that 20 people can get together and *prevent* the others from reconstructing the secret, no matter how many of them there are? As it turns out, we can [153].

The math is complicated, but the basic idea is that everyone gets two shares: a "yes" share and a "no" share. When it comes time to reconstruct the secret, people submit one of their shares. The actual share they submit depends on whether they wish the secret reconstructed. If there are m or more "yes" shares and fewer than n "no" shares, the secret can be reconstructed. Otherwise, it cannot.

Of course, nothing prevents a sufficient number of "yes" people from going off in a corner without the "no" people (assuming they know who they are) and reconstructing the secret. But in a situation where everyone submits their shares into a central computer, this scheme will work.

Secret Sharing with Disenrollment

You've set up your secret-sharing system and now you want to fire one of your shareholders. You could set up a new scheme without that person, but that's time-consuming. There are methods for coping with this system. They allow a new sharing scheme to be activated instantly once one of the participants becomes untrustworthy [1004].

3.8 CRYPTOGRAPHIC PROTECTION OF DATABASES

The membership database of an organization is a valuable commodity. On the one hand, you want to distribute the database to all members. You want them to com-

municate with one another, exchange ideas, and invite each other over for cucumber sandwiches. On the other hand, if you distribute the membership database to everyone, copies are bound to fall into the hands of insurance salesmen and other annoying purveyors of junk mail.

Cryptography can ameliorate this problem. We can encrypt the database so that it is easy to extract the address of a single person but hard to extract a mailing list of all the members.

The scheme, from [550,549], is straightforward. Choose a one-way hash function and a symmetric encryption algorithm. Each record of the database has two fields. The index field is the last name of the member, operated on by the one-way hash function. The data field is the full name and address of the member, encrypted using the last name as the key. Unless you know the last name, you can't decrypt the data field.

Searching a specific last name is easy. First, hash the last name and look for the hashed value in the index field of the database. If there is a match, then that last name is in the database. If there are several matches, then there are several people in the database with the last name. Finally, for each matching entry, decrypt the full name and address using the last name as the key.

In [550] the authors use this system to protect a dictionary of 6000 Spanish verbs. They report minimal performance degradation due to the encryption. Additional complications in [549] handle searches on multiple indexes, but the idea is the same. The primary problem with this system is that it's impossible to search for people when you don't know how to spell their name. You can try variant spellings until you find the correct one, but it isn't practical to scan through everyone whose name begins with "Sch" when looking for "Schneier."

This protection isn't perfect. It is possible for a particularly persistent insurance salesperson to reconstruct the membership database through brute-force by trying every possible last name. If he has a telephone database, he can use it as a list of possible last names. This might take a few weeks of dedicated number crunching, but it can be done. It makes his job harder and, in the world of junk mail, "harder" quickly becomes "too expensive."

Another approach, in [185], allows statistics to be compiled on encrypted data.

CHAPTER 18

One-Way Hash Functions

18.1 BACKGROUND

A one-way hash function, $H(M)$, operates on an arbitrary-length pre-image message, M . It returns a fixed-length value, h .

$$h = H(M), \text{ where } h \text{ is of length } m$$

Many functions can take an arbitrary-length input and return an output of fixed length, but one-way hash functions have additional characteristics that make them one-way [1065]:

Given M , it is easy to compute h .

Given h , it is hard to compute M such that $H(M) = h$.

Given M , it is hard to find another message, M' , such that $H(M) = H(M')$.

If Mallory could do the hard things, he would undermine the security of every protocol that uses the one-way hash function. The whole point of the one-way hash function is to provide a “fingerprint” of M that is unique. If Alice signed M by using a digital signature algorithm on $H(M)$, and Bob could produce M' , another message different from M where $H(M) = H(M')$, then Bob could claim that Alice signed M' .

In some applications, one-wayness is insufficient; we need an additional requirement called **collision-resistance**.

It is hard to find two random messages, M and M' , such that $H(M) = H(M')$.

Remember the birthday attack from Section 7.4? It is not based on finding another message M' , such that $H(M) = H(M')$, but based on finding two random messages, M and M' , such that $H(M) = H(M')$.

The following protocol, first described by Gideon Yuval [1635], shows how—if the previous requirement were not true—Alice could use the birthday attack to swindle Bob.

- (1) Alice prepares two versions of a contract: one is favorable to Bob; the other bankrupts him.
- (2) Alice makes several subtle changes to each document and calculates the hash value for each. (These changes could be things like: replacing SPACE with SPACE-BACKSPACE-SPACE, putting a space or two before a carriage return, and so on. By either making or not making a single change on each of 32 lines, Alice can easily generate 2^{32} different documents.)
- (3) Alice compares the hash values for each change in each of the two documents, looking for a pair that matches. (If the hash function only outputs a 64-bit value, she would usually find a matching pair with 2^{32} versions of each.) She reconstructs the two documents that hash to the same value.
- (4) Alice has Bob sign the version of the contract that is favorable to him, using a protocol in which he only signs the hash value.
- (5) At some time in the future, Alice substitutes the contract Bob signed with the one that he didn't. Now she can convince an adjudicator that Bob signed the other contract.

This is a big problem. (One moral is to always make a cosmetic change to any document you sign.)

Other similar attacks could be mounted assuming a successful birthday attack. For example, an adversary could send an automated control system (on a satellite, perhaps) random message strings with random signature strings. Eventually, one of those random messages will have a valid signature. The adversary would have no idea what the command would do, but if his only objective was to tamper with the satellite, this would do it.

Length of One-Way Hash Functions

Hash functions of 64 bits are just too small to survive a birthday attack. Most practical one-way hash functions produce 128-bit hashes. This forces anyone attempting the birthday attack to hash 2^{64} random documents to find two that hash to the same value, not enough for lasting security. NIST, in its Secure Hash Standard (SHS), uses a 160-bit hash value. This makes the birthday attack even harder, requiring 2^{80} random hashes.

The following method has been proposed to generate a longer hash value than a given hash function produces.

- (1) Generate the hash value of a message, using a one-way hash function listed in this book.

- (2) Prepend the hash value to the message.
- (3) Generate the hash value of the concatenation of the message and the hash value.
- (4) Create a larger hash value consisting of the hash value generated in step (1) concatenated with the hash value generated in step (3).
- (5) Repeat steps (1) through (3) as many times as you wish, concatenating as you go.

Although this method has never been proved to be either secure or insecure, various people have some serious reservations about it [1262,859].

Overview of One-Way Hash Functions

It's not easy to design a function that accepts an arbitrary-length input, let alone make it one-way. In the real world, one-way hash functions are built on the idea of a **compression function**. This one-way function outputs a hash value of length n given an input of some larger length m [1069,414]. The inputs to the compression function are a message block and the output of the previous blocks of text (see Figure 18.1). The output is the hash of all blocks up to that point. That is, the hash of block M_i is

$$h_i = f(M_i, h_{i-1})$$

This hash value, along with the next message block, becomes the next input to the compression function. The hash of the entire message is the hash of the last block.

The pre-image should contain some kind of binary representation of the length of the entire message. This technique overcomes a potential security problem resulting from messages with different lengths possibly hashing to the same value [1069,414]. This technique is sometimes called **MD-strengthening** [930].

Various researchers have theorized that if the compression function is secure, then this method of hashing an arbitrary-length pre-image is also secure—but nothing has been proved [1138,1070,414].

A lot has been written on the design of one-way hash functions. For more mathematical information, consult [1028,793,791,1138,1069,414,91,858,1264]. Bart Preneel's thesis [1262] is probably the most comprehensive treatment of one-way hash functions.

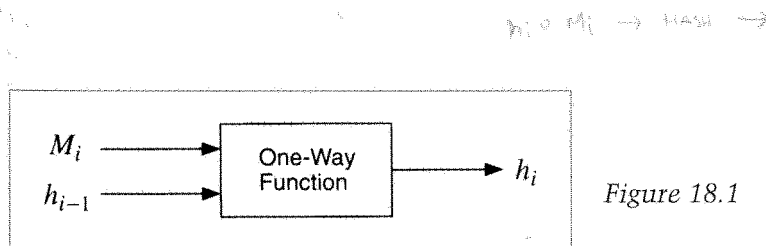


Figure 18.1 One-way function.

18.2 SNEFRU

Snefru is a one-way hash function designed by Ralph Merkle [1070]. (Snefru, like Khufu and Khafre, was an Egyptian pharaoh.) Snefru hashes arbitrary-length messages into either 128-bit or 256-bit values.

First the message is broken into chunks, each 512- m in length. (The variable m is the length of the hash value.) If the output is a 128-bit hash value, then the chunks are each 384 bits long; if the output is a 256-bit hash value, then the chunks are each 256 bits long.

The heart of the algorithm is function H , which hashes a 512-bit value into an m -bit value. The first m bits of H 's output are the hash of the block; the rest are discarded. The next block is appended to the hash of the previous block and hashed again. (The initial block is appended to a string of zeros.) After the last block (if the message isn't an even number of blocks long, zeros are used to pad the last block), the first m bits are appended to a binary representation of the length of the message and hashed one final time.

Function H is based on E , which is a reversible block-cipher function that operates on 512-bit blocks. H is the last m bits of the output of E XORed with the first m bits of the input of E .

The security of Snefru resides in function E , which randomizes data in several passes. Each pass is composed of 64 randomizing rounds. In each round a different byte of the data is used as an input to an S-box; the output word of the S-box is XORed with two neighboring words of the message. The S-boxes are constructed in a manner similar to those in Khafre (see Section 13.7). Some rotations are thrown in, too. Originally Snefru was designed with two passes.

Cryptanalysis of Snefru

Using differential cryptanalysis, Biham and Shamir demonstrated the insecurity of two-pass Snefru (128-bit hash value) [172]. Their attack finds pairs of messages that hash to the same value within minutes.

On 128-bit Snefru, their attacks work better than brute force for four passes or less. A birthday attack against Snefru takes 2^{64} operations; differential cryptanalysis can find a pair of messages that hash to the same value in $2^{28.5}$ operations for three-pass Snefru and $2^{44.5}$ operations for four-pass Snefru. Finding a message that hashes to a given value by brute force requires 2^{128} operations; differential cryptanalysis takes 2^{56} operations for three-pass Snefru and 2^{88} operations for four-pass Snefru.

Although Biham and Shamir didn't analyze 256-bit hash values, they extended their analysis to 224-bit hash values. Compared to a birthday attack that requires 2^{112} operations, they can find messages that hash to the same value in $2^{12.5}$ operations for two-pass Snefru, 2^{33} operations for three-pass Snefru, and 2^{81} operations for four-pass Snefru.

Currently, Merkle recommends using Snefru with at least eight passes [1073]. However, with this many passes the algorithm is significantly slower than either MD5 or SHA.

and 64-bit right half swapped; it is then XORed with a repeating one/zero pattern (128 bits worth), and then XORed with the current message block, M_i . This value then cascades into N ($N=8$ in the figures) processing stages. The other input to the processing stage is the previous hash value XORed with one of eight binary constant values.

One processing stage is given in Figure 18.3. The message block is broken into four 32-bit values. The previous hash value is also broken into four 32-bit values. The function f is given in Figure 18.4. Functions S_0 and S_1 are the same as they were in FEAL.

$$S_0(a,b) = \text{rotate left two bits } ((a + b) \bmod 256)$$

$$S_1(a,b) = \text{rotate left two bits } ((a + b + 1) \bmod 256)$$

The output of one processing stage becomes the input to the next processing stage. After the last processing stage, the output is XORed with the M_i and H_{i-1} , and then the next block is ready to be hashed.

Cryptanalysis of N-Hash

Bert den Boer discovered a way to produce collisions in the round function of N-Hash [1262]. Biham and Shamir used differential cryptanalysis to break 6-round

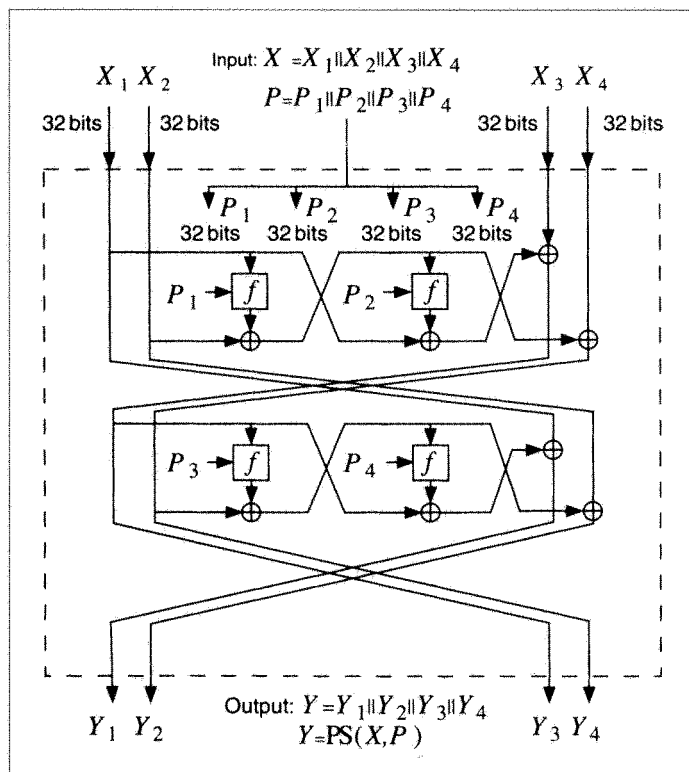
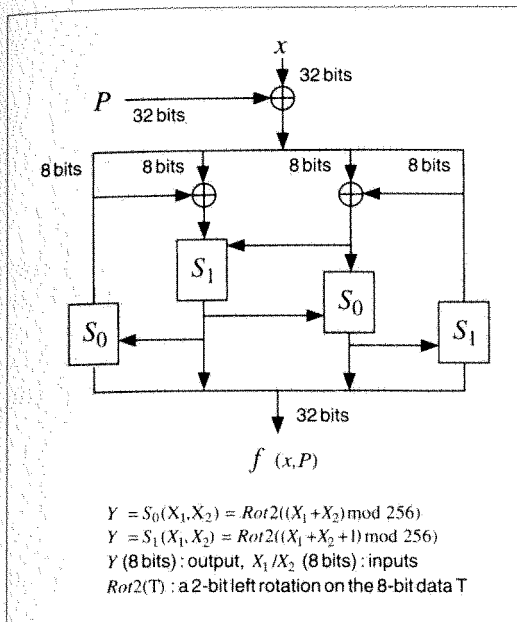


Figure 18.3 One processing stage of N-Hash.

Figure 18.4 Function f .

N -Hash [169,172]. Their particular attack (there certainly could be others) works for any N that is divisible by 3, and is more efficient than the birthday attack for any N less than 15.

The same attack can find pairs of messages that hash to the same value for 12-round N -Hash in 2^{56} operations, compared to 2^{64} operations for a brute-force attack. N -hash with 15 rounds is safe from differential cryptanalysis: The attack requires 2^{72} operations.

The algorithm's designers recommend using N -Hash with at least 8 rounds [1106]. Given the proven insecurity of N -Hash and FEAL (and its speed with 8 rounds), I recommend using another algorithm entirely.

18.4 MD4

MD4 is a one-way hash function designed by Ron Rivest [1318,1319,1321]. MD stands for **Message Digest**; the algorithm produces a 128-bit hash, or message digest, of the input message.

In [1319], Rivest outlined his design goals for the algorithm:

Security. It is computationally infeasible to find two messages that hashed to the same value. No attack is more efficient than brute force.

Direct Security. MD4's security is not based on any assumption, like the difficulty of factoring.

Speed. MD4 is suitable for high-speed software implementations. It is based on a simple set of bit manipulations on 32-bit operands.

Simplicity and Compactness. MD4 is as simple as possible, without large data structures or a complicated program.

Favor Little-Endian Architectures. MD4 is optimized for microprocessor architectures (specifically Intel microprocessors); larger and faster computers make any necessary translations.

After the algorithm was first introduced, Bert den Boer and Antoon Bosselaers successfully cryptanalyzed the last two of the algorithm's three rounds [202]. In an unrelated cryptanalytic result, Ralph Merkle successfully attacked the first two rounds [202]. Eli Biham discussed a differential cryptanalysis attack against the first two rounds of MD4 [159]. Even though these attacks could not be extended to the full algorithm, Rivest strengthened the algorithm. The result is MD5.

18.5 MD5

MD5 is an improved version of MD4 [1386,1322]. Although more complex than MD4, it is similar in design and also produces a 128-bit hash.

Description of MD5

After some initial processing, MD5 processes the input text in 512-bit blocks, divided into 16 32-bit sub-blocks. The output of the algorithm is a set of four 32-bit blocks, which concatenate to form a single 128-bit hash value.

First, the message is padded so that its length is just 64 bits short of being a multiple of 512. This padding is a single 1-bit added to the end of the message, followed by as many zeros as are required. Then, a 64-bit representation of the message's length (before padding bits were added) is appended to the result. These two steps serve to make the message length an exact multiple of 512 bits in length (required for the rest of the algorithm), while ensuring that different messages will not look the same after padding.

Four 32-bit variables are initialized:

$A = 0x01234567$

$B = 0x89abcdef$

$C = 0xfedcba98$

$D = 0x76543210$

These are called **chaining variables**.

Now, the main loop of the algorithm begins. This loop continues for as many 512-bit blocks as are in the message.

The four variables are copied into different variables: a gets A , b gets B , c gets C , and d gets D .

The main loop has four rounds (MD4 had only three rounds), all very similar. Each round uses a different operation 16 times. Each operation performs a nonlinear func-

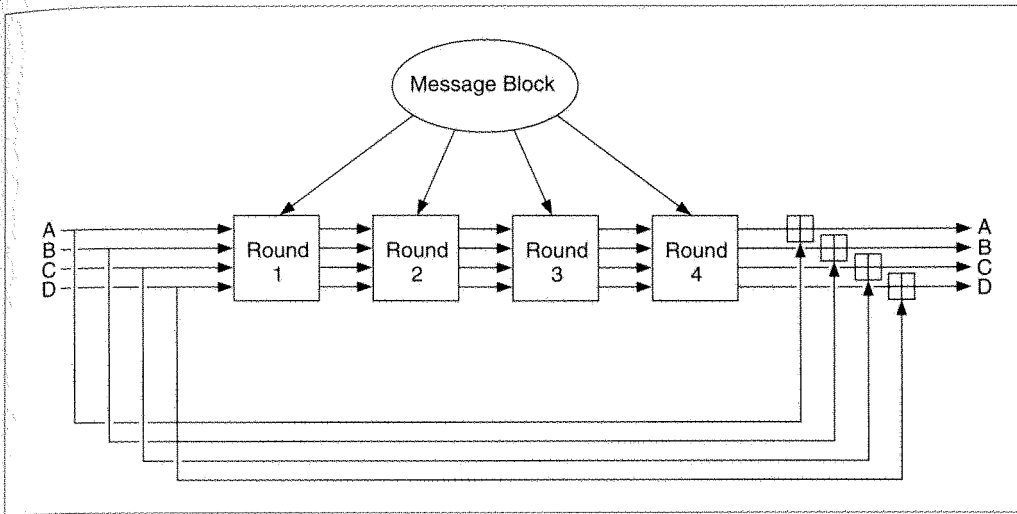


Figure 18.5 MD5 main loop.

tion on three of a , b , c , and d . Then it adds that result to the fourth variable, a sub-block of the text and a constant. Then it rotates that result to the right a variable number of bits and adds the result to one of a , b , c , or d . Finally the result replaces one of a , b , c , or d . See Figures 18.5 and 18.6.

There are four nonlinear functions, one used in each operation (a different one for each round).

$$F(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z)$$

$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge (\neg Z))$$

$$H(X,Y,Z) = X \oplus Y \oplus Z$$

$$I(X,Y,Z) = Y \oplus (X \vee (\neg Z))$$

($\oplus$ is XOR, $\wedge$ is AND, $\vee$ is OR, and $\neg$ is NOT.)

These functions are designed so that if the corresponding bits of X , Y , and Z are independent and unbiased, then each bit of the result will also be independent and unbiased. The function F is the bit-wise conditional: If X then Y else Z . The function H is the bit-wise parity operator.

If M_j represents the j th sub-block of the message (from 0 to 15), and $\lll s$ represents a left circular shift of s bits, the four operations are:

$$FF[a,b,c,d,M_j,s,t_i] \text{ denotes } a = b + ((a + F(b,c,d) + M_j + t_i) \lll s)$$

$$GG[a,b,c,d,M_j,s,t_i] \text{ denotes } a = b + ((a + G(b,c,d) + M_j + t_i) \lll s)$$

$$HH[a,b,c,d,M_j,s,t_i] \text{ denotes } a = b + ((a + H(b,c,d) + M_j + t_i) \lll s)$$

$$II[a,b,c,d,M_j,s,t_i] \text{ denotes } a = b + ((a + I(b,c,d) + M_j + t_i) \lll s)$$

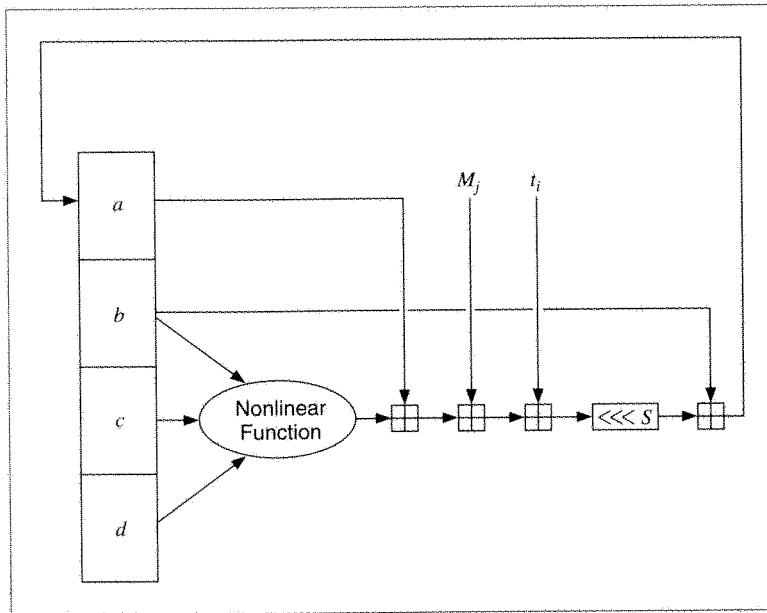


Figure 18.6 One MD5 operation.

The four rounds (64 steps) look like:

Round 1:

```

FF (a, b, c, d, M0, 7, 0xd76aa478)
FF (d, a, b, c, M1, 12, 0xe8c7b756)
FF (c, d, a, b, M2, 17, 0x242070db)
FF (b, c, d, a, M3, 22, 0xc1bdceee)
FF (a, b, c, d, M4, 7, 0xf57c0faf)
FF (d, a, b, c, M5, 12, 0x4787c62a)
FF (c, d, a, b, M6, 17, 0xa8304613)
FF (b, c, d, a, M7, 22, 0xfd469501)
FF (a, b, c, d, M8, 7, 0x698098d8)
FF (d, a, b, c, M9, 12, 0x8b44f7af)
FF (c, d, a, b, M10, 17, 0xffff5bb1)
FF (b, c, d, a, M11, 22, 0x895cd7be)
FF (a, b, c, d, M12, 7, 0x6b901122)
FF (d, a, b, c, M13, 12, 0xfd987193)
FF (c, d, a, b, M14, 17, 0xa679438e)
FF (b, c, d, a, M15, 22, 0x49b40821)

```

Round 2:

GG ($a, b, c, d, M_1, 5, 0xf61e2562$)
GG ($d, a, b, c, M_6, 9, 0xc040b340$)
GG ($c, d, a, b, M_{11}, 14, 0x265e5a51$)
GG ($b, c, d, a, M_0, 20, 0xe9b6c7aa$)
GG ($a, b, c, d, M_5, 5, 0xd62f105d$)
GG ($d, a, b, c, M_{10}, 9, 0x02441453$)
GG ($c, d, a, b, M_{15}, 14, 0xd8a1e681$)
GG ($b, c, d, a, M_4, 20, 0xe7d3fbc8$)
GG ($a, b, c, d, M_9, 5, 0x21e1cde6$)
GG ($d, a, b, c, M_{14}, 9, 0xc33707d6$)
GG ($c, d, a, b, M_3, 14, 0xf4d50d87$)
GG ($b, c, d, a, M_8, 20, 0x455a14ed$)
GG ($a, b, c, d, M_{13}, 5, 0xa9e3e905$)
GG ($d, a, b, c, M_2, 9, 0xfcefa3f8$)
GG ($c, d, a, b, M_7, 14, 0x676f02d9$)
GG ($b, c, d, a, M_{12}, 20, 0x8d2a4c8a$)

Round 3:

HH ($a, b, c, d, M_5, 4, 0xffffa3942$)
HH ($d, a, b, c, M_8, 11, 0x8771f681$)
HH ($c, d, a, b, M_{11}, 16, 0x6d9d6122$)
HH ($b, c, d, a, M_{14}, 23, 0xfde5380c$)
HH ($a, b, c, d, M_1, 4, 0xa4beea44$)
HH ($d, a, b, c, M_4, 11, 0x4bdecfa9$)
HH ($c, d, a, b, M_7, 16, 0xf6bb4b60$)
HH ($b, c, d, a, M_{10}, 23, 0xbebfbfc70$)
HH ($a, b, c, d, M_{13}, 4, 0x289b7ec6$)
HH ($d, a, b, c, M_0, 11, 0xeaa127fa$)
HH ($c, d, a, b, M_3, 16, 0xd4ef3085$)
HH ($b, c, d, a, M_6, 23, 0x04881d05$)
HH ($a, b, c, d, M_9, 4, 0xd9d4d039$)
HH ($d, a, b, c, M_{12}, 11, 0xe6db99e5$)
HH ($c, d, a, b, M_{15}, 16, 0x1fa27cf8$)
HH ($b, c, d, a, M_2, 23, 0xc4ac5665$)

Round 4:

$\Pi(a, b, c, d, M_0, 6, 0xf4292244)$
 $\Pi(d, a, b, c, M_7, 10, 0x432aff97)$
 $\Pi(c, d, a, b, M_{14}, 15, 0xab9423a7)$
 $\Pi(b, c, d, a, M_5, 21, 0xfc93a039)$
 $\Pi(a, b, c, d, M_{12}, 6, 0x655b59c3)$
 $\Pi(d, a, b, c, M_3, 10, 0x8f0ccc92)$
 $\Pi(c, d, a, b, M_{10}, 15, 0xffeff47d)$
 $\Pi(b, c, d, a, M_1, 21, 0x85845dd1)$
 $\Pi(a, b, c, d, M_8, 6, 0x6fa87e4f)$
 $\Pi(d, a, b, c, M_{15}, 10, 0xfe2ce6e0)$
 $\Pi(c, d, a, b, M_6, 15, 0xa3014314)$
 $\Pi(b, c, d, a, M_{13}, 21, 0x4e0811a1)$
 $\Pi(a, b, c, d, M_4, 6, 0xf7537e82)$
 $\Pi(d, a, b, c, M_{11}, 10, 0xbd3af235)$
 $\Pi(c, d, a, b, M_2, 15, 0x2ad7d2bb)$
 $\Pi(b, c, d, a, M_9, 21, 0xeb86d391)$

Those constants, t_i , were chosen as follows:

In step i , t_i is the integer part of $2^{32} \cdot \text{abs}(\sin(i))$, where i is in radians.

After all of this, a , b , c , and d are added to A , B , C , D , respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A , B , C , and D .

Security of MD5

Ron Rivest outlined the improvements of MD5 over MD4 [1322]:

1. A fourth round has been added.
2. Each step now has a unique additive constant.
3. The function G in round 2 was changed from $((X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z))$ to $((X \wedge Z) \vee (Y \neg (Z)))$ to make G less symmetric.
4. Each step now adds in the result of the previous step. This promotes a faster avalanche effect.
5. The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike.
6. The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds.

Tom Berson attempted to use differential cryptanalysis against a single round of MD5 [144], but his attack is ineffective against all four rounds. A more successful attack by den Boer and Bosselaers produces collisions using the compression function in MD5 [203,1331,1336]. This does not lend itself to attacks against MD5 in practical applications, and it does not affect the use of MD5 in Luby-Rackoff-like encryption algorithms (see Section 14.11). It does mean that one of the basic design principles of MD5—to design a collision-resistant compression function—has been violated. Although it is true that “there seems to be a weakness in the compression function, but it has no practical impact on the security of the hash function” [1336], I am wary of using MD5.

18.6 MD2

MD2 is another 128-bit one-way hash function designed by Ron Rivest [801,1335]. It, along with MD5, is used in the PEM protocols (see Section 24.10). The security of MD2 is dependent on a random permutation of bytes. This permutation is fixed, and depends on the digits of π . $S_0, S_1, S_2, \dots, S_{255}$ is the permutation. To hash a message M :

- (1) Pad the message with i bytes of value i so that the resulting message is a multiple of 16 bytes long.
- (2) Append a 16-byte checksum to the message.
- (3) Initialize a 48-byte block: $X_0, X_1, X_2, \dots, X_{47}$. Set the first 16 bytes of X to be 0, the second 16 bytes of X to be the first 16 bytes of the message, and the third 16 bytes of X to be the XOR of the first 16 bytes of X and the second 16 bytes of X .
- (4) This is the compression function:

$t = 0$
 For $j = 0$ to 17
 For $k = 0$ to 47
 $t = X_k \text{ XOR } S_t$
 $X_k = t$
 $t = (t + j) \bmod 256$
- (5) Set the second 16 bytes of X to be the second 16 bytes of the message, and the third 16 bytes of X to be the XOR of the first 16 bytes of X and the second 16 bytes of X . Do step (4). Repeat steps (5) and (4) with every 16 bytes of the message, in turn.
- (6) The output is the first 16 bytes of X .

Although no weaknesses in MD2 have been found (see [1262]), it is slower than most other suggested hash functions.

18.7 SECURE HASH ALGORITHM (SHA)

NIST, along with the NSA, designed the Secure Hash Algorithm (SHA) for use with the Digital Signature Standard (see Section 20.2) [1154]. (The standard is the Secure Hash Standard (SHS); SHA is the algorithm used in the standard.)

According to the *Federal Register* [539]:

A Federal Information Processing Standard (FIPS) for Secure Hash Standard (SHS) is being proposed. This proposed standard specified a Secure Hash Algorithm (SHA) for use with the proposed Digital Signature Standard . . . Additionally, for applications not requiring a digital signature, the SHA is to be used whenever a secure hash algorithm is required for Federal applications.

And

This Standard specifies a Secure Hash Algorithm (SHA), which is necessary to ensure the security of the Digital Signature Algorithm (DSA). When a message of any length $< 2^{64}$ bits is input, the SHA produces a 160-bit output called a message digest. The message digest is then input to the DSA, which computes the signature for the message. Signing the message digest rather than the message often improves the efficiency of the process, because the message digest is usually much smaller than the message. The same message digest should be obtained by the verifier of the signature when the received version of the message is used as input to SHA. The SHA is called secure because it is designed to be computationally infeasible to recover a message corresponding to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with a very high probability, result in a different message digest, and the signature will fail to verify. The SHA is based on principles similar to those used by Professor Ronald L. Rivest of MIT when designing the MD4 message digest algorithm [1319], and is closely modelled after that algorithm.

SHA produces a 160-bit hash, longer than MD5.

Description of SHA

First, the message is padded to make it a multiple of 512 bits long. Padding is exactly the same as in MD5: First append a one, then as many zeros as necessary to make it 64 bits short of a multiple of 512, and finally a 64-bit representation of the length of the message before padding.

Five 32-bit variables (MD5 has four variables, but this algorithm needs to produce a 160-bit hash) are initialized as follows:

$A = 0x67452301$

$B = 0xefcdab89$

$C = 0x98badcfe$

$D = 0x10325476$

$E = 0xc3d2e1f0$

The main loop of the algorithm then begins. It processes the message 512 bits at a time and continues for as many 512-bit blocks as are in the message.

First the five variables are copied into different variables: a gets A , b gets B , c gets C , d gets D , and e gets E .

The main loop has four rounds of 20 operations each (MD5 has four rounds of 16 operations each). Each operation performs a nonlinear function on three of a , b , c , d , and e , and then does shifting and adding similar to MD5.

SHA's set of nonlinear functions is:

$$f_t(X, Y, Z) = (X \wedge Y) \vee ((\neg X) \wedge Z), \text{ for } t = 0 \text{ to } 19. \text{ round 1}$$

$$f_t(X, Y, Z) = X \oplus Y \oplus Z, \text{ for } t = 20 \text{ to } 39. \text{ round 2}$$

$$f_t(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), \text{ for } t = 40 \text{ to } 59. \text{ round 3}$$

$$f_t(X, Y, Z) = X \oplus Y \oplus Z, \text{ for } t = 60 \text{ to } 79. \text{ round 4}$$

Four constants are used in the algorithm:

$$K_t = 0x5a827999, \text{ for } t = 0 \text{ to } 19.$$

$$K_t = 0x6ed9eba1, \text{ for } t = 20 \text{ to } 39.$$

$$K_t = 0x8f1bbcdc, \text{ for } t = 40 \text{ to } 59.$$

$$K_t = 0xca62c1d6, \text{ for } t = 60 \text{ to } 79.$$

(If you wonder where those numbers came from: $0x5a827999 = 2^{1/2}/4$, $0x6ed9eba1 = 3^{1/2}/4$, $0x8f1bbcdc = 5^{1/2}/4$, and $0xca62c1d6 = 10^{1/2}/4$.)

The message block is transformed from 16 32-bit words (M_0 to M_{15}) to 80 32-bit words (W_0 to W_{79}) using the following algorithm:

$$W_t = M_t, \text{ for } t = 0 \text{ to } 15$$

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1, \text{ for } t = 16 \text{ to } 79.$$

(As an interesting aside, the original SHA specification did not have the left circular shift. The change "corrects a technical flaw that made the standard less secure than had been thought" [543]. The NSA has refused to elaborate on the exact nature of the flaw.)

If t is the operation number (from 1 to 80), W_t represents the t th sub-block of the expanded message, and $\lll s$ represents a left circular shift of s bits, then the main loop looks like:

FOR $t = 0$ to 79

$$TEMP = (a \lll 5) + f_t(b, c, d) + e + W_t + K_t$$

$$e = d$$

$$d = c$$

$$c = b \lll 30$$

$$b = a$$

$$a = TEMP$$

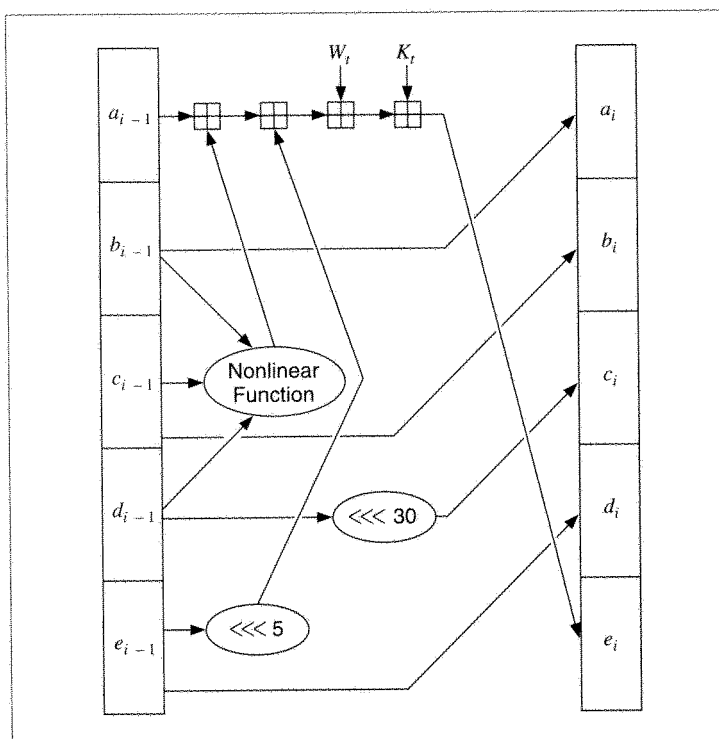


Figure 18.7 One SHA operation.

Figure 18.7 shows one operation. Shifting the variables accomplishes the same thing as MD5 does by using different variables in different locations.

After all of this, a , b , c , d , and e are added to A , B , C , D , and E respectively, and the algorithm continues with the next block of data. The final output is the concatenation of A , B , C , D , and E .

Security of SHA

SHA is very similar to MD4, but has a 160-bit hash value. The main changes are the addition of an expand transformation and the addition of the previous step's output into the next step for a faster avalanche effect. Ron Rivest made public the design decisions behind MD5, but SHA's designers did not. Here are Rivest's MD5 improvements to MD4 and how they compare with SHA's:

1. "A fourth round has been added." SHA does this, too. However, in SHA the fourth round uses the same f function as the second round.
2. "Each step now has a unique additive constant." SHA keeps the MD4 scheme where it reuses the constants for each group of 20 rounds.
3. "The function G in round 2 was changed from $((X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z))$ to $((X \wedge Z) \vee (Y \wedge \neg(Z)))$ to make G less symmetric." SHA uses the MD4 version: $((X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z))$.

4. "Each step now adds in the result of the previous step. This promotes a faster avalanche effect." This change has been made in SHA as well. The difference in SHA is that a fifth variable is added, and not b , c , or d , which is already used in f . This subtle change makes the den Boer-Bosselaers attack against MD5 impossible against SHA.
5. "The order in which message sub-blocks are accessed in rounds 2 and 3 is changed, to make these patterns less alike." SHA is completely different, since it uses a cyclic error-correcting code.
6. "The left circular shift amounts in each round have been approximately optimized, to yield a faster avalanche effect. The four shifts used in each round are different from the ones used in other rounds." SHA uses a constant shift amount in each round. This shift amount is relatively prime to the word size, as in MD4.

This leads to the following comparison: SHA is MD4 with the addition of an expand transformation, an extra round, and better avalanche effect; MD5 is MD4 with improved bit hashing, an extra round, and better avalanche effect.

There are no known cryptographic attacks against SHA. Because it produces a 160-bit hash, it is more resistant to brute-force attacks (including birthday attacks) than 128-bit hash functions covered in this chapter.

18.8 RIPE-MD

RIPE-MD was developed for the European Community's RIPE project [1305] (see Section 25.7). The algorithm is a variation of MD4, designed to resist known cryptanalytic attacks, and produce a 128-bit hash value. The rotations and the order of the message words are modified. Additionally, two instances of the algorithm, differing only in the constants, run in parallel. After each block, the output of both instances are added to the chaining variables. This seems to make the algorithm highly resistant to cryptanalysis.

18.9 HAVAL

HAVAL is a variable-length one-way hash function [1646]. It is a modification of MD5. HAVAL processes messages in blocks of 1024 bits, twice those of MD5. It has eight 32-bit chaining variables, twice those of MD5. It has a variable number of rounds, from three to five (each of which has 16 steps), and it can produce a hash length of 128, 160, 192, 224, or 256 bits.

HAVAL replaces MD5's simple nonlinear functions with highly nonlinear 7-variable functions, each of which satisfies the strict avalanche criterion. Each round uses a single function, but in every step a different permutation is applied to the inputs. It has a new message order and every step (except those in the first round) uses a different additive constant. The algorithm also has two rotations.

The core of the algorithm is

$$\begin{aligned} TEMP &= (f(j, A, B, C, D, E, F, G) \lll 7) + (H \lll 11) + M[i][r(j)] + K(j) \\ H &= G; G = F; F = E; E = D; D = C; C = B; B = A; A = TEMP \end{aligned}$$

The variable number of rounds and variable-length output mean there are 15 versions of this algorithm. Den Boer's and Bosselaers's attack against MD5 [203] does not apply to HAVAL because of the rotation of H .

18.10 OTHER ONE-WAY HASH FUNCTIONS

MD3 is yet another hash function designed by Ron Rivest. It had several flaws and never really made it out of the laboratory, although a description was recently published in [1335].

A group of researchers at the University of Waterloo have proposed a one-way hash function based on iterated exponentiation in $GF(2^{593})$ [22]. In this scheme, a message is divided into 593-bit blocks; beginning with the first block, the blocks are successively exponentiated. Each exponent is the result of the computation with the previous block; the first exponent is given by an IV.

Ivan Damgård designed a one-way hash function based on the knapsack problem (see Section 19.2) [414]; it can be broken in about 2^{32} operations [290,1232,787].

Steve Wolfram's cellular automata [1608] have been proposed as a basis for one-way hash functions. An early implementation [414] is insecure [1052,404]. Another one-way hash function, Cellhash [384,404], and an improved version, Subhash [384,402,405], are based on cellular automata; both are designed for hardware. Boogish mixes the design principles of Cellhash with those of MD4 [402,407]. StepRightUp can be implemented as a hash function as well [402].

Claus Schnorr proposed a one-way hash function based on the discrete Fourier transform, called FFT-Hash, in the summer of 1991 [1399]; it was broken a few months later by two independent groups [403,84]. Schnorr proposed a revised version, called FFT-Hash II (the previous version was renamed FFT-Hash I) [1400], which was broken a few weeks later [1567]. Schnorr has proposed further modifications [1402,1403] but, as it stands, the algorithm is much slower than the others in this chapter. Another hash function, called SL_2 [1526], is insecure [315].

Additional theoretical work on constructing one-way hash functions from one-way functions and one-way permutations can be found in [412,1138,1342].

18.11 ONE-WAY HASH FUNCTIONS USING SYMMETRIC BLOCK ALGORITHMS

It is possible to use a symmetric block cipher algorithm as a one-way hash function. The idea is that if the block algorithm is secure, then the one-way hash function will also be secure.

The most obvious method is to encrypt the message with the algorithm in CBC or CFB mode, a fixed key, and IV; the last ciphertext block is the hash value. These methods are described in various standards using DES: both modes in [1143], CBC in [1145], CBC in [55,56,54]. This just isn't good enough for one-way hash functions, although it will work for a MAC (see Section 18.14) [29].

A cleverer approach uses the message block as the key, the previous hash value as the input, and the current hash value as the output.

The actual hash functions proposed are even more complex. The block size is usually the key length, and the size of the hash value is the block size. Since most block algorithms are 64 bits, several schemes are designed around a hash that is twice the block size.

Assuming the hash function is correct, the security of the scheme is based on the security of the underlying block function. There are exceptions, though. Differential cryptanalysis is easier against block functions in hash functions than against block functions used for encryption: The key is known, so several tricks can be applied; only one right pair is needed for success; and you can generate as much chosen plaintext as you want. Some work on these lines is [1263,858,1313].

What follows is a summary of the various hash functions that have appeared in the literature [925,1465,1262]. Statements about attacks against these schemes assume that the underlying block cipher is secure; that is, the best attack against them is brute force.

One useful measure for hash functions based on block ciphers is the **hash rate**, or the number of n -bit messages blocks, where n is the block size of the algorithm, processed per encryption. The higher the hash rate, the faster the algorithm. (This measure was given the opposite definition in [1262], but the definition given here is more intuitive and is more widely used. This can be confusing.)

Schemes Where the Hash Length Equals the Block Size

The general scheme is as follows (see Figure 18.8):

$$H_0 = I_H, \text{ where } I_H \text{ is a random initial value}$$

$$H_i = E_A(B) \oplus C$$

where A , B , and C can be either M_i , H_{i-1} , $(M_i \oplus H_{i-1})$, or a constant (assumed to be 0). H_0 is some random initial value: I_H . The message is divided up into block-size chunks, M_i , and processed individually. And there is some kind of MD-strengthening, perhaps the same padding procedure used in MD5 and SHA.

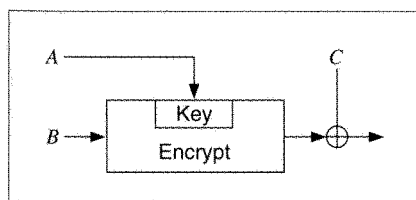


Figure 18.8 General hash function where the hash length equals the block size.

Table 18.1
Secure Hash Functions Where the
Block Length Equals the Hash Size

$H_i = E_{H_{i-1}}(M_i) \oplus M_i$
$H_i = E_{H_{i-1}}(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$
$H_i = E_{H_{i-1}}(M_i) \oplus H_{i-1} \oplus M_i$
$H_i = E_{H_{i-1}}(M_i \oplus H_{i-1}) \oplus M_i$
$H_i = E_{M_i}(H_{i-1}) \oplus H_{i-1}$
$H_i = E_{M_i}(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$
$H_i = E_{M_i}(H_{i-1}) \oplus M_i \oplus H_{i-1}$
$H_i = E_{M_i}(M_i \oplus H_{i-1}) \oplus H_{i-1}$
$H_i = E_{M_i \oplus H_{i-1}}(M_i) \oplus M_i$
$H_i = E_{M_i \oplus H_{i-1}}(H_{i-1}) \oplus H_{i-1}$
$H_i = E_{M_i \oplus H_{i-1}}(M_i) \oplus H_{i-1}$
$H_i = E_{M_i \oplus H_{i-1}}(H_{i-1}) \oplus M_i$

The three different variables can take on one of four possible values, so there are 64 total schemes of this type. Bart Preneel studied them all [1262].

Fifteen are trivially weak because the result does not depend on one of the inputs. Thirty-seven are insecure for more subtle reasons. Table 18.1 lists the 12 secure schemes remaining: The first 4 are secure against all attacks (see Figure 18.9) and the last 8 are secure against all but a fixed-point attack, which is not really worth worrying about.

The first scheme was described in [1028]. The third scheme was described in [1555,1105,1106] and was proposed as an ISO standard [766]. The fifth scheme was proposed by Carl Meyer, but is commonly called Davies-Meyer in the literature [1606,1607,434,1028]. The tenth scheme was proposed as a hash-function mode for LOKI [273].

The first, second, third, fourth, ninth, and eleventh schemes have a hash rate of 1; the key length equals the block length. The others have a rate of k/n , where k is the key length. This means that if the key length is shorter than the block length, then the message block can only be the length of the key. It is not recommended that the message block be longer than the key length, even if the encryption algorithm's key length is longer than the block length.

If the block algorithm has a DES-like complementation property and DES-like weak keys, there is an additional attack that is possible against all 12 schemes. The attack isn't very dangerous and not really worth worrying about. However, you can solve it by fixing bits 2 and 3 of the key to "01" or "10" [1081,1107]. Of course, this reduces the length of k from 56 bits to 54 bits (in DES, for example) and decreases the hash rate.

The following schemes, proposed in the literature, have been shown to be insecure.

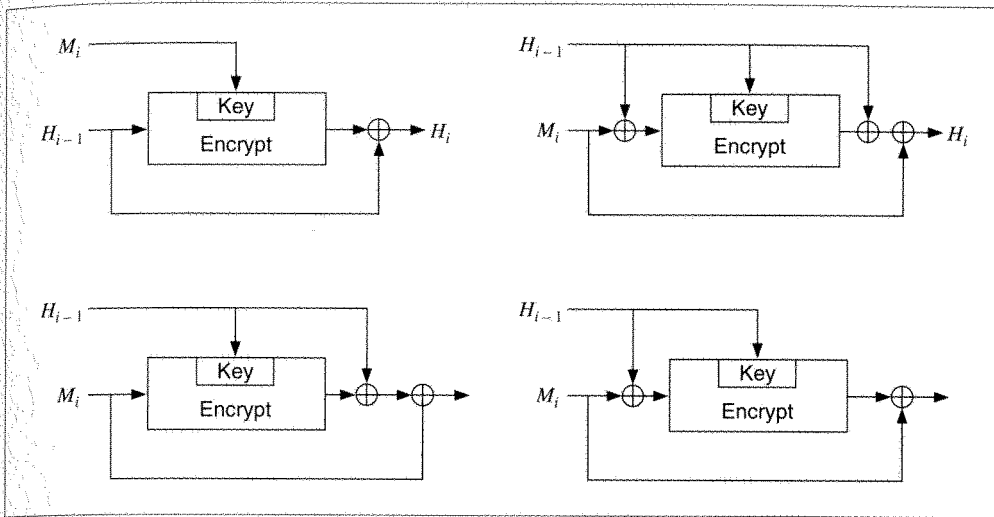


Figure 18.9 The four secure hash functions where the block length equals the hash size.

This scheme [1282] was broken in [369]:

$$H_i = E_{M_i}(H_{i-1})$$

Davies and Price proposed a variant which cycles the entire message through the algorithm twice [432,433]. Coppersmith's attack works on this variant with not much larger computational requirements [369].

Another scheme [432,458] was shown insecure in [1606]:

$$H_i = E_{M_i \oplus H_{i-1}}(H_{i-1})$$

This scheme was shown insecure in [1028] (c is a constant):

$$H_i = E_c(M_i \oplus H_{i-1}) \oplus M_i \oplus H_{i-1}$$

Modified Davies-Meyer

Lai and Massey modified the Davies-Meyer technique to work with the IDEA cipher [930,925]. IDEA has a 64-bit block size and 128-bit key size. Their scheme is

$$H_0 = I_H \text{ where } I_H \text{ is a random initial value}$$

$$H_i = E_{H_{i-1}, M_i}(H_{i-1})$$

This function hashes the message in blocks of 64 bits and produces a 64-bit hash value (See Figure 18.10).

No known attack on this scheme is easier than brute force.

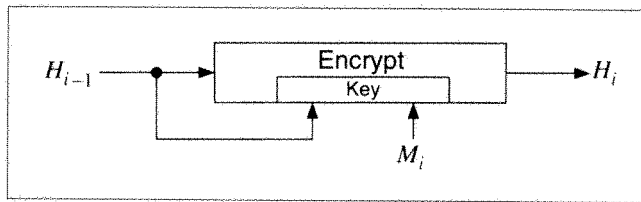


Figure 18.10 Modified Davies-Meyer.

Preneel-Bosselaers-Govaerts-Vandewalle

This hash function, first proposed in [1266], produces a hash value twice the block length of the encryption algorithm: A 64-bit algorithm produces a 128-bit hash.

With a 64-bit block algorithm, the scheme produces two 64-bit hash values, G_i and H_i , which are concatenated to produce the 128-bit hash. With most block algorithms, the block size is 64 bits. Two adjacent message blocks, L_i and R_i , each the size of the block length, are hashed together.

$$G_0 = I_G, \text{ where } I_G \text{ is a random initial value}$$

$$H_0 = I_H, \text{ where } I_H \text{ is another random initial value}$$

$$G_i = E_{L_i \oplus H_{i-1}}(R_i \oplus G_{i-1}) \oplus R_i \oplus G_{i-1} \oplus H_{i-1}$$

$$H_i = E_{L_i \oplus R_i}(H_{i-1} \oplus G_{i-1}) \oplus L_i \oplus G_{i-1} \oplus H_{i-1}$$

Lai demonstrates attacks against this scheme that, in some instances, make the birthday attack trivially solvable [925,926]. Preneel [1262] and Coppersmith [372] also have successful attacks against this scheme. Do not use it.

Quisquater-Girault

This scheme, first proposed in [1279], generates a hash that is twice the block length and has a hash rate of 1. It has two hash values, G_i and H_i , and two blocks, L_i and R_i , are hashed together.

$$G_0 = I_G, \text{ where } I_G \text{ is a random initial value}$$

$$H_0 = I_H, \text{ where } I_H \text{ is another random initial value}$$

$$W_i = E_{L_i}(G_{i-1} \oplus R_i) \oplus R_i \oplus H_{i-1}$$

$$G_i = E_{R_i}(W_i \oplus L_i) \oplus G_{i-1} \oplus H_{i-1} \oplus L_i$$

$$H_i = W_i \oplus G_{i-1}$$

This scheme appeared in a 1989 draft ISO standard [764], but was dropped in a later version [765]. Security problems with this scheme were identified in [1107,925, 1262,372]. (Actually, the version in the proceedings was strengthened after the version presented at the conference was attacked.) In some instances the birthday attack is solvable with a complexity of 2^{39} , not 2^{64} , through brute force. Do not use this scheme.

LOKI Double-Block

This algorithm is a modification of Quisquater-Girault, specifically designed to work with LOKI [273]. All parameters are as in Quisquater-Girault.

$G_0 = I_G$, where I_G is a random initial value

$H_0 = I_H$, where I_H is another random initial value

$W_i = E_{L_i \oplus G_{i-1}}(G_{i-1} \oplus R_i) \oplus R_i \oplus H_{i-1}$

$G_i = E_{R_i \oplus H_{i-1}}(W_i \oplus L_i) \oplus G_{i-1} \oplus H_{i-1} \oplus L_i$

$H_i = W_i \oplus G_{i-1}$

Again, in some instances the birthday attack is trivially solvable [925,926,1262,372,736]. Do not use this scheme.

Parallel Davies-Meyer

This is yet another attempt at an algorithm with a hash rate of 1 that produces a hash twice the block length [736].

$G_0 = I_G$, where I_G is a random initial value

$H_0 = I_H$, where I_H is another random initial value

$G_i = E_{L_i \oplus R_i}(G_{i-1} \oplus L_i) \oplus L_i \oplus H_{i-1}$

$H_i = E_{L_i}(H_{i-1} \oplus R_i) \oplus R_i \oplus H_{i-1}$

Unfortunately, this scheme isn't secure either [928,861]. As it turns out, a double-length hash function with a hash rate of 1 cannot be more secure than Davies-Meyer [861].

Tandem and Abreast Davies-Meyer

Another way around the inherent limitations of a block cipher with a 64-bit key uses an algorithm, like IDEA (see Section 13.9), with a 64-bit block and a 128-bit key. These two schemes produce a 128-bit hash value and have a hash rate of $\frac{1}{2}$ [930,925].

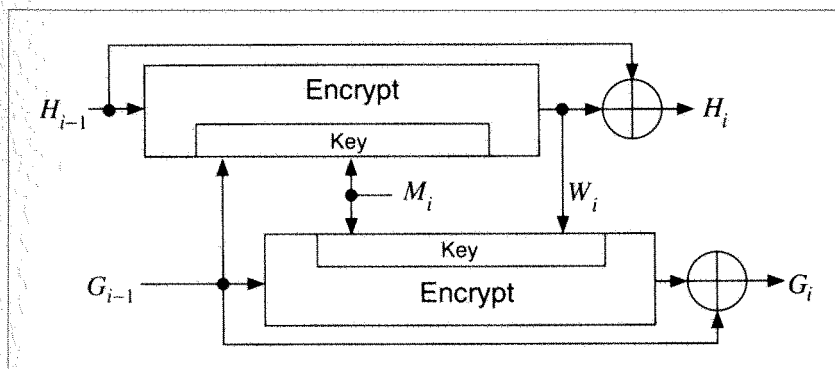


Figure 18.11 Tandem Davies-Meyer.

In this first scheme, two modified Davies-Meyer functions work in tandem (see Figure 18.11).

$G_0 = I_G$, where I_G is some random initial value

$H_0 = I_H$, where I_H is some other random initial value

$W_i = E_{G_{i-1} \parallel M_i}(H_{i-1})$

$G_i = G_{i-1} \oplus E_{M_i \parallel W_i}(G_{i-1})$

$H_i = W_i \oplus H_{i-1}$

The following scheme uses two modified Davies-Meyer functions side-by-side (see Figure 18.12).

$G_0 = I_G$, where I_G is some random initial value

$H_0 = I_H$, where I_H is some other random initial value

$G_i = G_{i-1} \oplus E_{M_i \parallel H_{i-1}}(\neg G_{i-1})$

$H_i = H_{i-1} \oplus E_{G_{i-1} \parallel M_i}(H_{i-1})$

In both schemes, the two 64-bit hash values G_i and H_i are concatenated to produce a single 128-bit hash.

As far as anyone knows, these algorithms have ideal security for a 128-bit hash function: Finding a message that hashes to a given hash value requires 2^{128} attempts, and finding two random messages that hash to the same value requires 2^{64} attempts—assuming that there is no better way to attack the block algorithm than by using brute force.

MDC-2 and MDC-4

MDC-2 and MDC-4 were first developed at IBM [1081,1079]. MDC-2, sometimes called Meyer-Schilling, is under consideration as an ANSI and ISO standard [61,765]; a variant was proposed in [762]. MDC-4 is specified for the RIPE project [1305] (see Section 25.7). The specifications use DES as the block function, although in theory any encryption algorithm could be used.

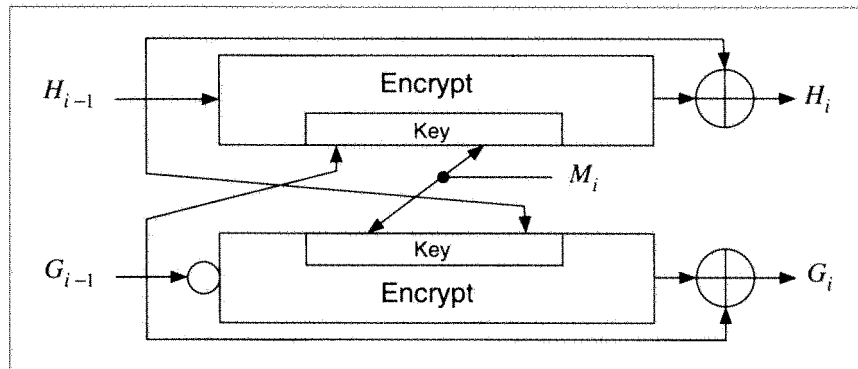


Figure 18.12 Abreast Davies-Meyer.

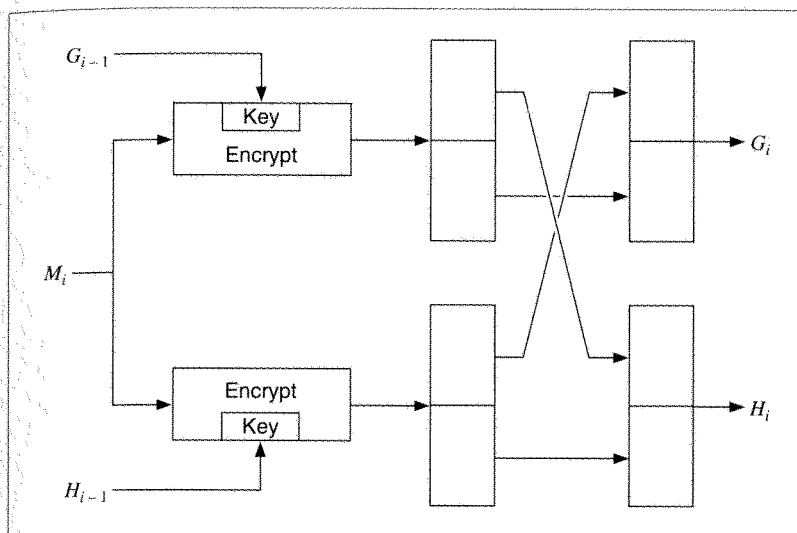


Figure 18.13 MDC-2.

MDC-2 has a hash rate of $\frac{1}{2}$, and produces a hash value twice the length of the block size. It is shown in Figure 18.13. MDC-4 also produces a hash value twice the length of the block size, and has a hash rate of $\frac{1}{4}$ (see Figure 18.14).

These schemes have been analyzed in [925,1262]. They are secure against current computing power, but they are not nearly as secure as the designers have estimated. If the block algorithm is DES, they have been looked at with respect to differential cryptanalysis [1262].

Both MDC-2 and MDC-4 are patented [223].

AR Hash Function

The AR hash function was developed by Algorithmic Research, Ltd. and has been distributed by the ISO for information purposes only [767]. Its basic structure is a variant of the underlying block cipher (DES in the reference) in CBC mode. The last two ciphertext blocks and a constant are XORed to the current message block and encrypted by the algorithm. The hash is the last two ciphertext blocks computed. The message is processed twice, with two different keys, so the hash function has a hash rate of $\frac{1}{2}$. The first key is 0x0000000000000000, the second key is 0x2a41522f4446502a, and c is 0x0123456789abcdef. The result is compressed to a single 128-bit hash value. See [750] for the details.

$$H_i = E_K(M_i \oplus H_{i-1} \oplus H_{i-2} \oplus c) \oplus M_i$$

This sounds interesting, but it is insecure. After considerable preprocessing, it is possible to find collisions for this hash function easily [416].

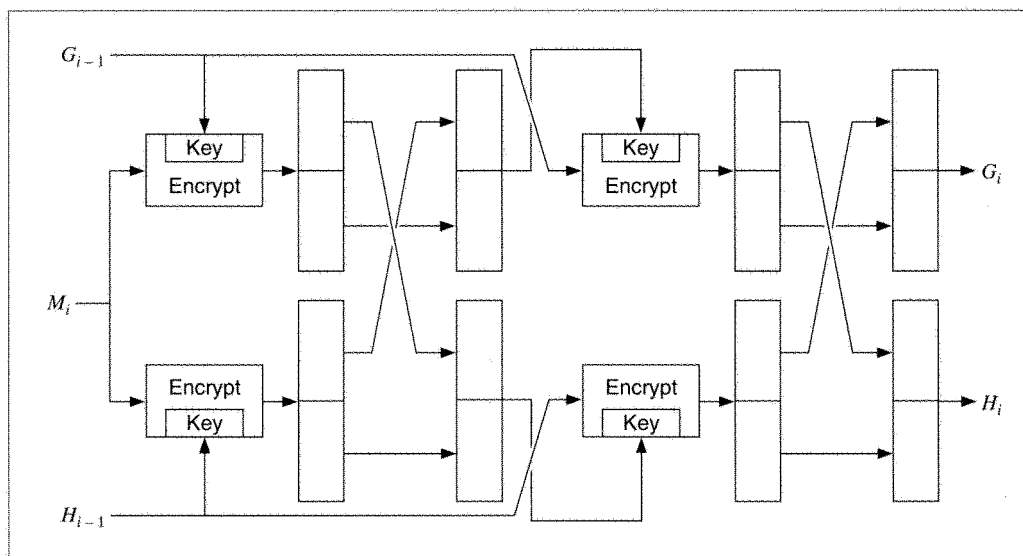


Figure 18.14 MDC-4.

GOST Hash Function

This hash function comes from Russia, and is specified in the standard GOST R 34.11-94 [657]. It uses the GOST block algorithm (see Section 14.1), although in theory it could use any block algorithm with a 64-bit block size and a 256-bit key. The function produces a 256-bit hash value.

The compression function, $H_i = f(M_i, H_{i-1})$ (both operands are 256-bit quantities) is defined as follows:

- (1) Generate four GOST encryption keys by some linear mixing of M_i , H_{i-1} , and some constants.
- (2) Use each key to encrypt a different 64 bits of H_{i-1} in ECB mode. Store the resulting 256 bits into a temporary variable, S .
- (3) H_i is a complex, although linear, function of S , M_i , and H_{i-1} .

The final hash of M is not the hash of the last block. There are actually three chaining variables: H_n is the hash of the last message block, Z is the XOR of all the message blocks, and L is the length of the message. Given those variables and the padded last block, M' , the final hash value is:

$$H = f(Z \oplus M', f(L, f(M', H_n)))$$

The documentation is a bit confusing (and in Russian), but I think all that is correct. In any case, this hash function is specified for use with the Russian Digital Signature Standard (see Section 20.3).

Other Schemes

Ralph Merkle proposed a scheme using DES, but it's slow; it only processes seven message bits per iteration and each iteration involves two DES encryptions [1065, 1069]. Another scheme [1642, 1645] is insecure [1267]; it was once proposed as an ISO standard.

18.12 USING PUBLIC-KEY ALGORITHMS

It is possible to use a public-key encryption algorithm in a block chaining mode as a one-way hash function. If you then throw away the private key, breaking the hash would be as difficult as reading the message without the private key.

Here's an example using RSA. If M is the message to be hashed, n is the product of two primes p and q , and e is another large number relatively prime to $(p-1)(q-1)$, then the hash function, $H(M)$, would be

$$H(M) = M^e \bmod n$$

An even easier solution would be to use a single strong prime as the modulus p . Then:

$$H(M) = M^e \bmod p$$

Breaking this problem is probably as difficult as finding the discrete logarithm of e . The problem with this algorithm is that it's far slower than any others discussed here. I don't recommend it for that reason.

18.13 CHOOSING A ONE-WAY HASH FUNCTION

The contenders seem to be SHA, MD5, and constructions based on block ciphers; the others really haven't been studied enough to be in the running. I vote for SHA. It has a longer hash value than MD5, is faster than the various block-cipher constructions, and was developed by the NSA. I trust the NSA's abilities at cryptanalysis, even if they don't make their results public.

Table 18.2 gives timing measurements for some hash functions. They are meant for comparison purposes only.

18.14 MESSAGE AUTHENTICATION CODES

A message authentication code, or MAC, is a key-dependent one-way hash function. MACs have the same properties as the one-way hash functions discussed previously, but they also include a key. (This does not imply that you can publish the key to a MAC and use it as a one-way hash function.) Only someone with the identical key can verify the hash. They are very useful to provide authenticity without secrecy.

MACs can be used to authenticate files between users. They can also be used by a single user to determine if his files have been altered, perhaps by a virus. A user

Table 18.2
Encryption Speeds of Some Hash Functions on a 33 MHz 486SX

Algorithm	Hash Length	Encryption Speed (kilobytes/second)
Abreast Davies-Meyer (with IDEA)	128	22
Davies-Meyer (with DES)	64	9
GOST Hash	256	11
HAVAL (3 passes)	variable	168
HAVAL (4 passes)	variable	118
HAVAL (5 passes)	variable	95
MD2	128	23
MD4	128	236
MD5	128	174
N-HASH (12 rounds)	128	29
N-HASH (15 rounds)	128	24
RIPE-MD	128	182
SHA	160	75
Snerfu (4 passes)	128	48
Snerfu (8 passes)	128	23

could compute the MAC of his files and store that value in a table. If the user used instead a one-way hash function, then the virus could compute the new hash value after infection and replace the table entry. A virus could not do that with a MAC, because the virus does not know the key.

An easy way to turn a one-way hash function into a MAC is to encrypt the hash value with a symmetric algorithm. Any MAC can be turned into a one-way hash function by making the key public.

CBC-MAC

The simplest way to make a key-dependent one-way hash function is to encrypt a message with a block algorithm in CBC or CFB modes. The hash is the last encrypted block, encrypted once more in CBC or CFB modes. The CBC method is specified in ANSI X9.9 [54], ANSI X9.19 [56], ISO 8731-1 [759], ISO 9797 [763], and an Australian standard [1496]. Differential cryptanalysis can break this scheme with reduced-round DES or FEAL as the underlying block algorithms [1197].

The potential security problem with this method is that the receiver must have the key, and that key allows him to generate messages with the same hash value as a given message by decrypting in the reverse direction.

Message Authenticator Algorithm (MAA)

This algorithm is an ISO standard [760]. It produces a 32-bit hash, and was designed for mainframe computers with a fast multiply instruction [428].

$$v = v \lll 1$$

$$e = v \oplus w$$

$$x = (((e + y) \bmod 2^{32}) \vee A \wedge C) * (x \oplus M_i) \bmod 2^{32} - 1$$

$$y = (((e + x) \bmod 2^{32}) \vee B \wedge D) * (y \oplus M_i) \bmod 2^{32} - 2$$

Iterate these for each message block, M_i , and the resultant hash is the XOR of x and y . The variables v and e are determined from the key. A , B , C , and D are constants.

This algorithm is probably in wide use, but I can't believe it is all that secure. It was designed a long time ago, and isn't very complicated.

Bidirectional MAC

This MAC produces a hash value twice the length of the block algorithm [978]. First, compute the CBC-MAC of the message. Then, compute the CBC-MAC of the message with the blocks in reverse order. The bidirectional MAC value is simply the concatenation of the two. Unfortunately, this construction is insecure [1097].

Jueneman's Methods

This MAC is also called a quadratic congruential manipulation detection code (QCMDC) [792,789]. First, divide the message into m -bit blocks. Then:

$$H_0 = I_H, \text{ where } I_H \text{ is the secret key}$$

$$H_i = (H_{i-1} + M_i)^2 \bmod p, \text{ where } p \text{ is a prime less than } 2^m - 1$$

and $+$ denotes integer addition

Jueneman suggests $n = 16$ and $p = 2^{31} - 1$. In [792] he also suggests that an additional key be used as H_1 , with the actual message starting at H_2 .

Because of a variety of birthday-type attacks discovered in conjunction with Don Coppersmith, Jueneman suggested computing the QCMDC four times, using the result of one iteration as the IV for the next iteration, and then concatenating the results to obtain a 128-bit hash value [793]. This was further strengthened by doing the four iterations in parallel and cross-linking them [790,791]. This scheme was broken by Coppersmith [376].

Another variant [432,434] replaced the addition operation with an XOR and used message blocks significantly smaller than p . H_0 was also set, making it a keyless one-way hash function. After this scheme was attacked [612], it was strengthened as part of the European Open Shop Information-TeleTrust project [1221], quoted in CCITT X.509 [304], and adopted in ISO 10118 [764,765]. Unfortunately, Coppersmith has broken this scheme as well [376]. There has been some research using exponents other than 2 [603], but none of it has been promising.

RIPE-MAC

RIPE-MAC was invented by Bart Preneel [1262] and adopted by the RIPE project [1305] (see Section 18.8). It is based on ISO 9797 [763], and uses DES as a block encryption function. RIPE-MAC has two flavors: one using normal DES, called

RIPE-MAC1, and another using triple-DES for even greater security, called RIPE-MAC3. RIPE-MAC1 uses one DES encryption per 64-bit message block; RIPE-MAC3 uses three.

The algorithm consists of three parts. First, the message is expanded to a length that is a multiple of 64 bits. Next, the expanded message is divided up into 64-bit blocks. A keyed compression function is used to hash these blocks, under the control of a secret key, into a single block of 64 bits. This is the step that uses either DES or triple-DES. Finally, the output of this compression is subjected to another DES-based encryption with a different key, derived from the key used in the compression. See [1305] for details.

IBC-Hash

IBC-Hash is another MAC adopted by the RIPE project [1305] (see Section 18.8). It is interesting because it is provably secure; the chance of successful attack can be quantified. Unfortunately, every message must be hashed with a different key. The chosen level of security puts constraints on the maximum message size that can be hashed—something no other function in this chapter does. Given these considerations, the RIPE report recommends that IBC-Hash be used only for long, infrequently sent messages.

The heart of the function is

$$h_i = ((M_i \bmod p) + v) \bmod 2^n$$

The secret key is the pair p and v , where p is an n -bit prime and v is a random number less than 2^n . The M_i values are derived by a carefully specified padding procedure. The probabilities of breaking both the one-wayness and the collision-resistance can be quantified, and users can choose their security level by changing the parameters.

One-Way Hash Function MAC

A one-way hash function can also be used as a MAC [1537]. Assume Alice and Bob share a key K , and Alice wants to send Bob a MAC for message M . Alice concatenates K and M , and computes the one-way hash of the concatenation: $H(K, M)$. This hash is the MAC. Since Bob knows K , he can reproduce Alice's result. Mallory, who does not know K , can't.

This method works with MD-strengthening techniques, but has serious problems. Mallory can always add new blocks to the end of the message and compute a valid MAC. This attack can be thwarted if you put the message length at the beginning, but Preneel is suspicious of this scheme [1265]. It is better to put the key at the end of the message, $H(M, K)$, but this has some problems as well [1265]. If H is one-way but not collision-free, Mallory can forge messages. Still better is $H(K, M, K)$, or $H(K_1, M, K_2)$, where K_1 and K_2 are different [1537]. Preneel is still suspicious [1265].

The following constructions seem secure:

$$H(K_1, H(K_2, M))$$

$$H(K, H(K, M))$$

$$H(K, p, M, K), \text{ where } p \text{ pads } K \text{ to a full message block.}$$

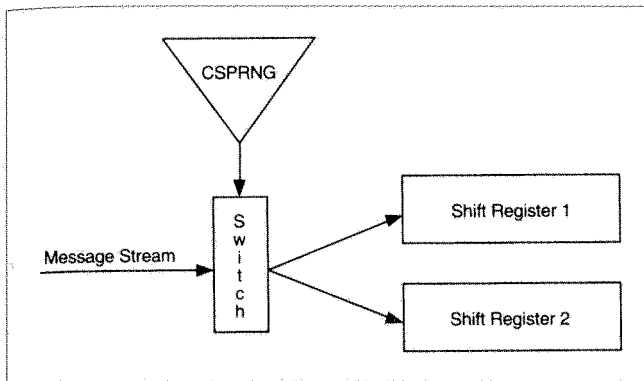


Figure 18.15 Stream cipher MAC.

The best approach is to concatenate at least 64 bits of the key with each message block. This makes the one-way hash function less efficient, because the message blocks are smaller, but it is much more secure [1265].

Alternatively, use a one-way hash function and a symmetric algorithm. Hash the file, then encrypt the hash. This is more secure than first encrypting the file and then hashing the encrypted file, but it is vulnerable to the same attack as the $H(M, K)$ approach [1265].

Stream Cipher MAC

This MAC scheme uses stream ciphers (see Figure 18.15) [932]. A cryptographically secure pseudo-random-bit generator demultiplexes the message stream into two substreams. If the output bit of the bit generator k_i is 1, then the current message bit m_i is routed to the first substream; if the k_i is 0, the m_i is routed to the second substream. The substreams are each fed into a different LFSR (see Section 16.2). The output of the MAC is simply the final states of the shift registers.

Unfortunately, this method is not secure against small changes in the message [1523]. For example, if you alter the last bit of the message, then only 2 bits in the corresponding MAC value need to be altered to create a fake MAC; this can be done with reasonable probability. The author presents a more secure, and more complicated, alternative.