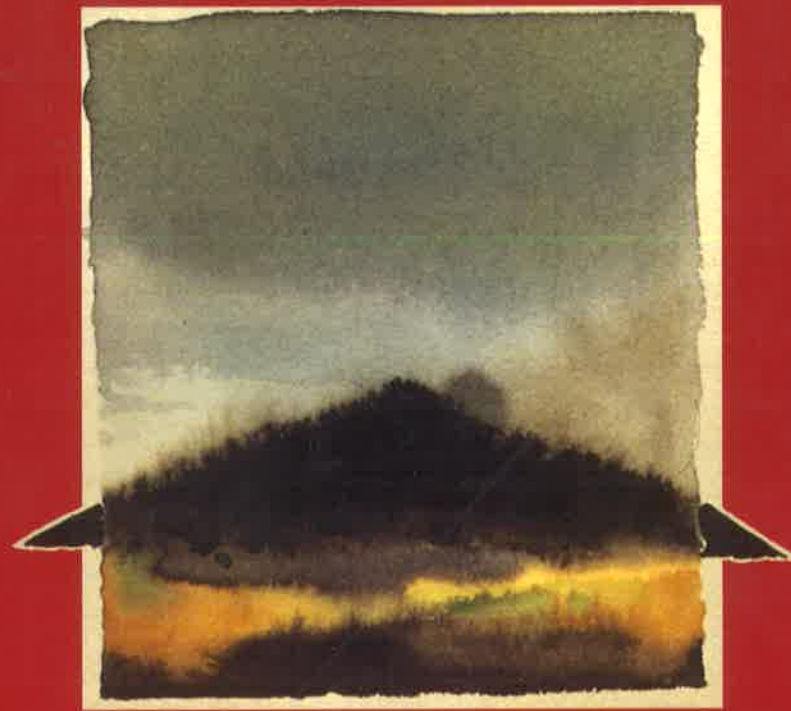


*"... the best introduction
to cryptography I've
ever seen.... The book
the National Security
Agency wanted never
to be published...."*

—Wired Magazine

**SECOND
EDITION**

APPLIED CRYPTOGRAPHY



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

BRUCE SCHNEIER

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
101 E. Minnekaka Parkway
Minneapolis, MN 55419

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

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 KEYSACES 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 AUTOMATON 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 513

- 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 527

- 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 SCI.CRYPT 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 675

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.

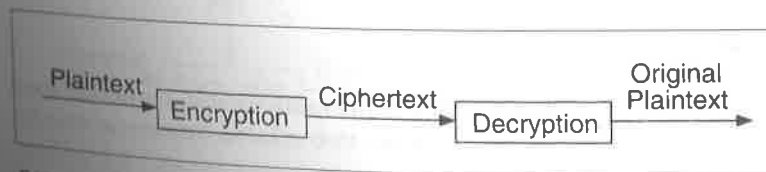


Figure 1.1 Encryption and Decryption.

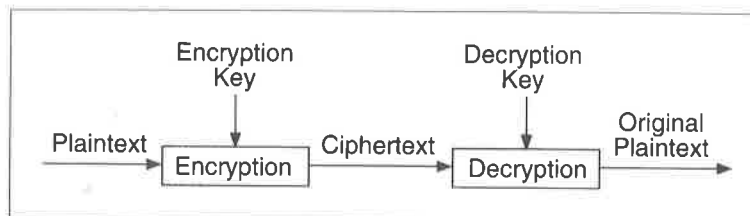


Figure 1.3 Encryption and decryption with two different keys.

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

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]?

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 timestamps, 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

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

The problem with this key management system is that Bob has to trust his security officer not to misuse everyone's keys. Even more significantly, all the employees have to trust the security officer not to misuse their keys. A far better solution is to use a secret-sharing protocol (see Section 3.7).

When Alice generates a key, she also divides up that key into some number of pieces. She then sends each piece—encrypted, of course—to a different company officer. None of those pieces alone is the key, but someone can gather all the pieces together and reconstruct the key. Now Alice is protected against any one malicious person, and Bob is protected against losing all of Alice's data after her run-in with the truck. Or, she could just store the different pieces, encrypted with each of the officer's different public keys, on her own hard disk. That way, no one gets involved with key management until it becomes necessary.

Another backup scheme [188] uses smart cards (see Section 24.13) for the temporary escrow of keys. Alice can put the key to secure her hard drive onto the smart card and give it to Bob while she is away. Bob can use the card to get into Alice's hard drive, but because the key is stored in the card Bob cannot learn it. And the system is bilaterally auditable: Bob can verify that the key will open Alice's drive, and when Alice returns she can verify if Bob has used the key and how many times.

Such a scheme makes no sense for data transmission. On a secure telephone, the key should exist for the length of the call and no longer. For data storage, as just described, key escrow can be a good idea. I've lost about one key every five years, and my memory is better than most. If 200 million people were using cryptography, that same rate would equal 40 million lost keys per year. I keep copies of my house keys with a neighbor because I may lose mine. If house keys were like cryptographic keys, and I lost them, I could never get inside and recover my possessions, ever again. Just as I keep off-site backups of my data, it makes sense to keep backups of my data-encryption keys.

8.9 COMPROMISED KEYS

All of the protocols, techniques, and algorithms in this book are secure only if the key (the private key in a public-key system) remains secret. If Alice's key is lost, stolen, printed in the newspaper, or otherwise compromised, then all her security is gone.

If the compromised key was for a symmetric cryptosystem, Alice has to change her key and hope the actual damage was minimal. If it was a private key, she has bigger problems; her public key is probably on servers all over the network. And if Eve gets access to Alice's private key, she can impersonate her on the network: reading encrypted mail, signing correspondence, entering into contracts, and so forth. Eve can, effectively, become Alice.

It is vital that news of a private key's compromise propagate quickly throughout the network. Any databases of public keys must immediately be notified that a particular private key has been compromised, lest some unsuspecting person encrypt a message in that compromised key.

One hopes Alice knows when her key was compromised. If a KDC is managing the keys, Alice should notify it that her key has been compromised. If there is no KDC, then she should notify all correspondents who might receive messages from her. Someone should publicize the fact that any message received after her key was lost is suspect, and that no one should send messages to Alice with the associated public key. The application should be using some sort of timestamp, and then users can determine which messages are legitimate and which are suspect.

If Alice doesn't know exactly when her key was compromised, things are more difficult. Alice may want to back out of a contract because the person who stole the key signed it instead of her. If the system allows this, then anyone can back out of a contract by claiming that his key was compromised before it was signed. It has to be a matter for an adjudicator to decide.

This is a serious problem and brings to light the dangers of Alice tying all of her identity to a single key. It would be better for Alice to have different keys for different applications—just as she has different physical keys in her pocket for different locks. Other solutions to this problem involve biometrics, limits on what can be done with a key, time delays, and countersigning.

These procedures and tips are hardly optimal, but are the best we can do. The moral of the story is to protect keys, and protect private keys above all else.

8.10 LIFETIME OF KEYS

No encryption key should be used for an indefinite period. It should expire automatically like passports and licenses. There are several reasons for this:

- The longer a key is used, the greater the chance that it will be compromised. People write keys down; people lose them. Accidents happen. If you use the same key for a year, there's a far greater chance of compromise than if you use it for a day.
- The longer a key is used, the greater the loss if the key is compromised. If a key is used only to encrypt a single budgetary document on a file server, then the loss of the key means only the compromise of that document. If the same key is used to encrypt all the budgetary information on the file server, then its loss is much more devastating.
- The longer a key is used, the greater the temptation for someone to spend the effort necessary to break it—even if that effort is a brute-force attack. Breaking a key shared between two military units for a day would enable someone to read and fabricate messages between those units for that day. Breaking a key shared by an entire military command structure for a year would enable that same person to read and fabricate messages throughout the world for a year. In our budget-conscious, post-Cold War world, which key would you choose to attack?

- It is generally easier to do cryptanalysis with more ciphertext encrypted with the same key.

For any cryptographic application, there must be a policy that determines the permitted lifetime of a key. Different keys may have different lifetimes. For a connection-based system, like a telephone, it makes sense to use a key for the length of the telephone call and to use a new one with each call.

Systems on dedicated communications channels are not as obvious. Keys should have relatively short lifetimes, depending on the value of the data and the amount of data encrypted during a given period. The key for a gigabit-per-second communications link might have to be changed more often than the key for a 9600-baud modem link. Assuming there is an efficient method of transferring new keys, session keys should be changed at least daily.

Key-encryption keys don't have to be replaced as frequently. They are used only occasionally (roughly once per day) for key exchange. This generates little ciphertext for a cryptanalyst to work with, and the corresponding plaintext has no particular form. However, if a key-encryption key is compromised, the potential loss is extreme: all communications encrypted with every key encrypted with the key-encryption key. In some applications, key-encryption keys are replaced only once a month or once a year. You have to balance the inherent danger in keeping a key around for a while with the inherent danger in distributing a new one.

Encryption keys used to encrypt data files for storage cannot be changed often. The files may sit encrypted on disk for months or years before someone needs them again. Decrypting them and re-encrypting them with a new key every day doesn't enhance security in any way; it just gives a cryptanalyst more to work with. One solution might be to encrypt each file with a unique file key, and then encrypt all the file keys with a key-encryption key. The key-encryption key should then be either memorized or stored in a secure location, perhaps in a safe somewhere. Of course, losing this key would mean losing all the individual file keys.

Private keys for public-key cryptography applications have varying lifetimes, depending on the application. Private keys used for digital signatures and proofs of identity may have to last years (even a lifetime). Private keys used for coin-flipping protocols can be discarded immediately after the protocol is completed. Even if a key's security is expected to last a lifetime, it may be prudent to change the key every couple of years. The private keys in many networks are good only for two years; after that the user must get a new private key. The old key would still have to remain secret, in case the user needed to verify a signature from that period. But the new key would be used to sign new documents, reducing the number of signed documents a cryptanalyst would have for an attack.

8.11 DESTROYING KEYS

Given that keys must be replaced regularly, old keys must be destroyed. Old keys are valuable, even if they are never used again. With them, an adversary can read old messages encrypted with those keys [65].

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 a . 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

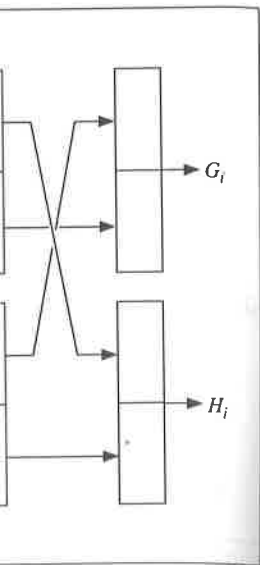
The contenders seem to be SHA, MD5, and constructions based on block ciphers; the others really haven't been studied enough to be in the running. I vote for SHA. It has a longer hash value than MD5, is faster than the various block-cipher constructions, and was developed by the NSA. I trust the NSA's 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. 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 could compute the MAC of his files and store that value in a table. If the user used



the standard GOST R
14.1), although in the
and a 256-bit key. The

e 256-bit quantities) is

ar mixing of M_i , H_{i-1} ,

n ECB mode. Store the

and H_{i-1} .

ere are actually three
is the sum mod 2^{256} of
ven those variables and

I think all that is cor-
he Russian Digital Sig-

Table 18.2
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
SNEFRU (4 passes)	128	48
SNEFRU (8 passes)	128	23

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].

Patents

The Diffie-Hellman key-exchange algorithm is patented in the United States [718] and Canada [719]. A group called Public Key Partners (PKP) licenses the patent, along with other public-key cryptography patents (see Section 25.5). The U.S. patent will expire on April 29, 1997.

22.2 STATION-TO-STATION PROTOCOL

Diffie-Hellman key exchange is vulnerable to a man-in-the-middle attack. One way to prevent this problem is to have Alice and Bob sign their messages to each other [500].

This protocol assumes that Alice has a certificate with Bob's public key and that Bob has a certificate with Alice's public key. These certificates have been signed by some trusted authority outside this protocol. Here's how Alice and Bob generate a secret key, k .

- (1) Alice generates a random number, x , and sends it to Bob.
- (2) Bob generates a random number, y . Using the Diffie-Hellman protocol he computes their shared key based on x and y : k . He signs x and y , and encrypts the signature using k . He then sends that, along with y , to Alice.

$$y, E_k(S_B(x, y))$$

- (3) Alice also computes k . She decrypts the rest of Bob's message and verifies his signature. Then she sends Bob a signed message consisting of x and y , encrypted in their shared key.

$$E_k(S_A(x, y))$$

- (4) Bob decrypts the message and verifies Alice's signature.

22.3 SHAMIR'S THREE-PASS PROTOCOL

This protocol, invented by Adi Shamir but never published, enables Alice and Bob to communicate securely without any advance exchange of either secret keys or public keys [1008].

This assumes the existence of a symmetric cipher that is commutative, that is:

$$E_A(E_B(P)) = E_B(E_A(P))$$

Alice's secret key is A ; Bob's secret key is B . Alice wants to send a message, M , to Bob. Here's the protocol.

- (1) Alice encrypts M with her key and sends Bob

$$C_1 = E_A(M)$$

- (2) Bob encrypts C_1 with his key and sends Alice

$$C_2 = E_B(E_A(M))$$

- (3) Alice decrypts C_2 with her key and sends Bob

$$C_3 = D_A(E_B(E_A(M))) = D_A(E_A(E_B(M))) = E_B(M)$$

- (4) Bob decrypts C_3 with his key to recover M .

One-time pads are commutative and have perfect secrecy, but they will not work with this protocol. With a one-time pad, the three ciphertext messages would be:

$$C_1 = P \oplus A$$

$$C_2 = P \oplus A \oplus B$$

$$C_3 = P \oplus B$$

Eve, who can record the three messages as they pass between Alice and Bob, simply XORs them together to retrieve the message:

$$C_1 \oplus C_2 \oplus C_3 = (P \oplus A) \oplus (P \oplus A \oplus B) \oplus (P \oplus B) = P$$

This clearly won't work.

Shamir (and independently, Jim Omura) described an encryption algorithm that will work with this protocol, one similar to RSA. Let p be a large prime for which $p - 1$ has a large prime factor. Choose an encryption key, e , such that e is relatively prime to $p - 1$. Calculate d such that $de \equiv 1 \pmod{p - 1}$.

To encrypt a message, calculate

$$C = M^e \pmod{p}$$

To decrypt a message, calculate

$$M = C^d \pmod{p}$$

There seems to be no way for Eve to recover M without solving the discrete logarithm problem, but this has never been proved.

Like Diffie-Hellman, this protocol allows Alice to initiate secure communication with Bob without knowing any of his keys. For Alice to use a public-key algorithm, she has to know his public key. With Shamir's three-pass protocol, she just sends him a ciphertext message. The same thing with a public-key algorithm looks like:

- (1) Alice asks Bob (or a KDC) for his public key.
- (2) Bob (or the KDC) sends Alice his public key.
- (3) Alice encrypts M with Bob's public key and sends it to Bob.

Shamir's three-pass protocol will fall to a man-in-the-middle attack.

22.4 COMSET

COMSET (COMMunications SETup) is a mutual identification and key exchange protocol developed for the RIPE project [1305] (see Section 25.7). Using public-key

technology and is used to establish pair-wise traffic encryption keys." Both this description and testimony submitted to the U.S. Congress by Lee Neuwirth of Cylink [1164] suggest a combination of key exchange and certificates similar to that used in the ISDN secure phone and it is plausible that FIREFLY too is based on exponentiation.

STU-IIIs are manufactured by AT&T and GE. Somewhere between 300,000 and 400,000 have been fielded through 1994. A new version, the Secure Terminal Equipment (STE), will work on ISDN lines.

24.5 KERBEROS

Kerberos is a trusted third-party authentication protocol designed for TCP/IP networks. A Kerberos service, sitting on the network, acts as a trusted arbitrator. Kerberos provides secure network authentication, allowing a person to access different machines on the network. Kerberos is based on symmetric cryptography (DES as implemented, but other algorithms could be used instead). Kerberos shares a different secret key with every entity on the network and knowledge of that secret key equals proof of identity.

Kerberos was originally developed at MIT for Project Athena. The Kerberos model is based on Needham-Schroeder's trusted third-party protocol (see Section 3.3) [1159]. The original version of Kerberos, Version 4, is specified in [1094,1499]. (Versions 1 through 3 were internal development versions.) Version 5, modified from Version 4, is specified in [876,877,878]. The best overview of Kerberos is [1163]. Other survey articles are [1384,1493], and two good articles on using Kerberos in the real world are [781,782].

The Kerberos Model

The basic Kerberos protocol was outlined in Section 3.3. In the Kerberos model there are entities—clients and servers—sitting on the network. Clients can be users, but can also be independent software programs that need to do things: download files, send messages, access databases, access printers, obtain administrative privileges, whatever.

Kerberos keeps a database of clients and their secret keys. For a human user, the secret key is an encrypted password. Network services requiring authentication, as well as clients who wish to use these services, register their secret key with Kerberos.

Because Kerberos knows everyone's secret key, it can create messages that convince one entity of another entity's identity. Kerberos also creates session keys which are given to a client and a server (or to two clients) and no one else. A session key is used to encrypt messages between the two parties, after which it is destroyed.

Kerberos uses DES for encryption. Kerberos Version 4 provided a nonstandard mode for authentication. This mode is weak: It fails to detect certain changes to the ciphertext (see Section 9.10). Kerberos Version 5 uses CBC mode.