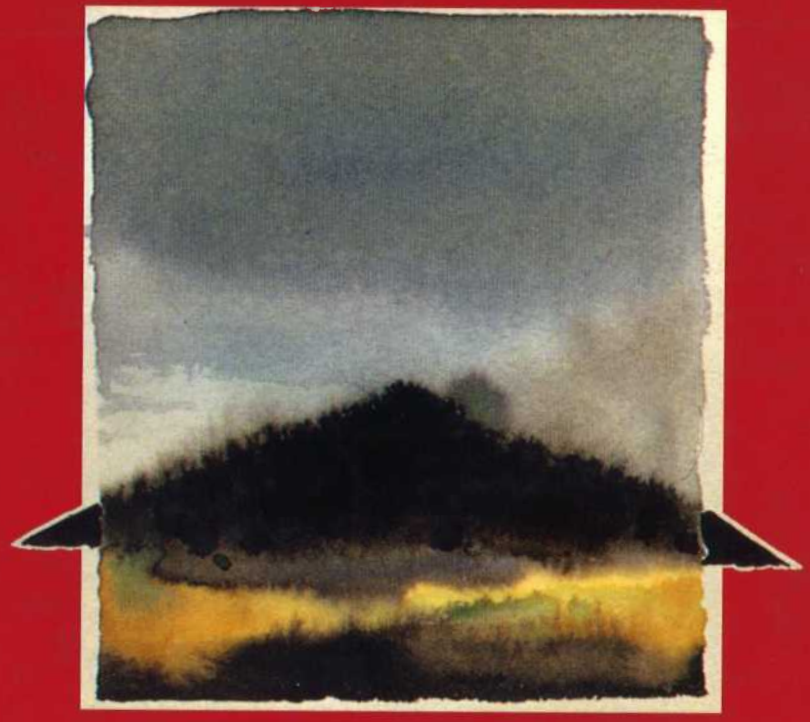
"... the best introduction to cryptography I've ever seen.... The book the National Security Agency wanted never to be published...."

—Wired *Magazine*

SECOND EDITION

APPLIED CRYPTOGRAPHY



Protocols, Algorithms, and Source Code in C

BRUCE SCHNEIER

ADOBE – Exhibit 1012, page 1

APPLIED CRYPTOGRAPHY, SECOND EDITION

PROTOCOLS, ALGORITHMS, AND SOURCE CODE IN C

BRUCE SCHNEIER



John Wiley & Sons, Inc.
New York • Chichester • Brisbane • Toronto • Singapore

ADOBE – Exhibit 1012, page 2

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 3 2 1

ADOBE – Exhibit 1012, page 3

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

ADOBE – Exhibit 1012, page 4

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*

ADOBE – Exhibit 1012, page 5

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*

ADOBE – Exhibit 1012, page 6

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*

ADOBE – Exhibit 1012, page 7

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*

ADOBE – Exhibit 1012, page 8

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*

ADOBE – Exhibit 1012, page 9

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~~ 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*

ADOBE – Exhibit 1012, page 10

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) *582*

ADOBE – Exhibit 1012, page 11

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 ~~679~~ 675

ADOBE – Exhibit 1012, page 12

CHAPTER 8

Key Management

Alice and Bob have a secure communications system. They play mental poker, simultaneously sign contracts, even exchange digital cash. Their protocols are secure. Their algorithms are top-notch. Unfortunately, they buy their keys from Eve's "Keys-R-Us," whose slogan is "You can trust us: Security is the middle name of someone our ex-mother-in-law's travel agent met at the Kwik-E-Mart."

Eve doesn't have to break the algorithms. She doesn't have to rely on subtle flaws in the protocols. She can use their keys to read all of Alice's and Bob's message traffic without lifting a cryptanalytic finger.

In the real world, key management is the hardest part of cryptography. Designing secure cryptographic algorithms and protocols isn't easy, but you can rely on a large body of academic research. Keeping the keys secret is much harder.

Cryptanalysts often attack both symmetric and public-key cryptosystems through their key management. Why should Eve bother going through all the trouble of trying to break the cryptographic algorithm if she can recover the key because of sloppy key storage procedures? Why should she spend $10 million building a cryptanalysis machine if she can spend $1000 bribing a clerk? Spending a million dollars to buy a well-placed communications clerk in a diplomatic embassy can be a bargain. The Walkers sold U.S. Navy encryption keys to the Soviets for years. The CIA's director of counterintelligence went for less than $2 million, wife included. That's far cheaper than building massive cracking machines and hiring brilliant cryptanalysts. Eve can steal the keys. She can arrest or abduct someone who knows the keys. She can seduce someone and get the keys that way. (The Marines who guarded the U.S. Embassy in Moscow were not immune to that attack.) It's a whole lot easier to find flaws in people than it is to find them in cryptosystems.

Alice and Bob must protect their key to the same degree as all the data it encrypts. If a key isn't changed regularly, this can be an enormous amount of data. Unfortunately, many commercial products simply proclaim "We use DES" and forget about everything else. The results are not very impressive.

ADOBE – Exhibit 1012, page 13

For example, the DiskLock program for Macintosh (version 2.1), sold at most software stores, claims the security of DES encryption. It encrypts files using DES. Its implementation of the DES algorithm is correct. However, DiskLock stores the DES key with the encrypted file. If you know where to look for the key, and want to read a file encrypted with DiskLock's DES, recover the key from the encrypted file and then decrypt the file. It doesn't matter that this program uses DES encryption—the implementation is completely insecure.

Further information on key management can be found in [457,98,1273,1225, 775,357]. The following sections discuss some of the issues and solutions.

8.1 GENERATING KEYS

The security of an algorithm rests in the key. If you're using a cryptographically weak process to generate keys, then your whole system is weak. Eve need not cryptanalyze your encryption algorithm; she can cryptanalyze your key generation algorithm.

Reduced Keyspaces

DES has a 56-bit key. Implemented properly, any 56-bit string can be the key; there are 2^{56} (10^{16}) possible keys. Norton Discreet for MS-DOS (versions 8.0 and earlier) only allows ASCII keys, forcing the high-order bit of each byte to be zero. The program also converts lowercase letters to uppercase (so the fifth bit of each byte is always the opposite of the sixth bit) and ignores the low-order bit of each byte, resulting in only 2^{40} possible keys. These poor key generation procedures have made its DES ten thousand times easier to break than a proper implementation.

Table 8.1 gives the number of possible keys with various constraints on the input strings. Table 8.2 gives the time required for an exhaustive search through all of those keys, given a million attempts per second. Remember, there is very little time differential between an exhaustive search for 8-byte keys and an exhaustive search of 4-, 5-, 6-, 7-, and 8-byte keys.

All specialized brute-force hardware and parallel implementations will work here. Testing a million keys per second (either with one machine or with multiple machines in parallel), it is feasible to crack lowercase-letter and lowercase-letter-

Table 8.1
Number of Possible Keys of Various Keyspaces

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	460,000	$1.2*10^{7}$	$3.1*10^{8}$	$8.0*10^{9}$	$2.1*10^{11}$
Lowercase letters and digits (36):	1,700,000	$6.0*10^{7}$	$2.2*10^{9}$	$7.8*10^{10}$	$2.8*10^{12}$
Alphanumeric characters (62):	$1.5*10^{7}$	$9.2*10^{8}$	$5.7*10^{10}$	$3.5*10^{12}$	$2.2*10^{14}$
Printable characters (95):	$8.1*10^{7}$	$7.7*10^{9}$	$7.4*10^{11}$	$7.0*10^{13}$	$6.6*10^{15}$
ASCII characters (128):	$2.7*10^{8}$	$3.4*10^{10}$	$4.4*10^{12}$	$5.6*10^{14}$	$7.2*10^{16}$
8-bit ASCII characters (256):	$4.3*10^{9}$	$1.1*10^{12}$	$2.8*10^{14}$	$7.2*10^{16}$	$1.8*10^{19}$

ADOBE – Exhibit 1012, page 14

Table 8.2
Exhaustive Search of Various Keyspaces (assume one million attempts per second)

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	.5 seconds	12 seconds	5 minutes	2.2 hours	2.4 days
Lowercase letters and digits (36):	1.7 seconds	1 minute	36 minutes	22 hours	33 days
Alphanumeric characters (62):	15 seconds	15 minutes	16 hours	41 days	6.9 years
Printable characters (95):	1.4 minutes	2.1 hours	8.5 days	2.2 years	210 years
ASCII characters (128):	4.5 minutes	9.5 hours	51 days	18 years	2300 years
8-bit ASCII characters (256):	1.2 hours	13 days	8.9 years	2300 years	580,000 years

and-number keys up to 8 bytes long, alphanumeric-character keys up to 7 bytes long, printable character and ASCII-character keys up to 6 bytes long, and 8-bit-ASCII-character keys up to 5 bytes long.

And remember, computing power doubles every 18 months. If you expect your keys to stand up against brute-force attacks for 10 years, you'd better plan accordingly.

Poor Key Choices

When people choose their own keys, they generally choose poor ones. They're far more likely to choose "Barney" than "*9 (hH/A." This is not always due to poor security practices; "Barney" is easier to remember than "*9 (hH/A." The world's most secure algorithm won't help much if the users habitually choose their spouse's names for keys or write their keys on little pieces of paper in their wallets. A smart brute-force attack doesn't try all possible keys in numerical order; it tries the obvious keys first.

This is called a **dictionary attack**, because the attacker uses a dictionary of common keys. Daniel Klein was able to crack 40 percent of the passwords on the average computer using this system [847,848]. No, he didn't try one password after another, trying to login. He copied the encrypted password file and mounted the attack offline. Here's what he tried:

1. The user's name, initials, account name, and other relevant ~~person~~ personal information as a possible password. All in all, up to 130 different passwords were tried based on this information. For an account name **klone** with a user named "Daniel V. Klein," some of the passwords that would be tried were: klone, klone0, klone1, klone123, dvk, dvkdvk, dklein, DKlein leinad, nielk, dvklein, danielk, DvkkvD, DANIEL-KLEIN, (klone), KleinD, and so on.
2. Words from various databases. These included lists of men's and women's names (some 16,000 in all); places (including variations so that "spain," "spanish," and "spaniard" would all be considered); names of famous people; cartoons and cartoon characters; titles, characters, and locations from films and science fiction stories; mythical creatures (garnered from *Bullfinch's Mythology* and dictionaries of mythical beasts); sports (includ-

ADOBE – Exhibit 1012, page 15

ing team names, nicknames, and specialized terms); numbers (both as numerals—"2001," and written out—"twelve"); strings of letters and numbers ("a," "aa," "aaa," "aaaa," etc.); Chinese syllables (from the Pinyin Romanization of Chinese, an international standard system of writing Chinese on an English keyboard); the King James Bible; biological terms; colloquial and vulgar phrases (such as "fuckyou," "ibmsux," and "deadhead"); keyboard patterns (such as "qwerty," "asdf," and "zxcvbn"); abbreviations (such as "roygbiv"—the colors in the rainbow, and "ooottafagvah"—a mnemonic for remembering the 12 cranial nerves); machine names (acquired from */etc/hosts*); characters, plays, and locations from Shakespeare; common Yiddish words; the names of asteroids; and a collection of words from various technical papers Klein previously published. All told, more than 60,000 separate words were considered per user (with any inter- and intra-dictionary duplicates being discarded).

3. Variations on the words from step 2. This included making the first letter uppercase or a control character, making the entire word uppercase, reversing the word (with and without the aforementioned capitalization), changing the letter 'o' to the digit '0' (so that the word "scholar" would also be checked as "sch0lar"), changing the letter 'l' to the digit '1' (so that the word "scholar" would also be checked as "scho1ar"), and performing similar manipulation to change the letter 'z' into the digit '2', and the letter 's' into the digit '5'. Another test was to make the word into a plural (irrespective of whether the word was actually a noun), with enough intelligence built in so that "dress" became "dresses," "house" became "houses," and "daisy" became "daisies." Klein did not consider pluralization rules exclusively, though, so that "datum" forgivably became "datums" (not "data"), while "sphynx" became "sphynxs" (and not "sphynges"). Similarly, the suffixes "-ed," "-er," and "-ing" were added to transform words like "phase" into "phased," "phaser," and "phasing." These additional tests added another 1,000,000 words to the list of possible passwords that were tested for each user.
4. Various capitalization variations on the words from step 2 that were not considered in step 3. This included all single-letter capitalization variations (so that "michael" would also be checked as "mIchael," "miChael," "micHael," "michAel," etc.), double-letter capitalization variations ("MIchael," "MiChael," "MicHael," . . . , "mIChael," "mIcHael," etc.), triple-letter variations, etc. The single-letter variations added roughly another 400,000 words to be checked per user, while the double-letter variations added another 1,500,000 words. Three-letter variations would have added at least another 3,000,000 words per user had there been enough time to complete the tests. Tests of four-, five-, and six-letter variations were deemed to be impracticable without much more computational horsepower to carry them out.
5. Foreign language words on foreign users. The specific test that was performed was to try Chinese language passwords on users with Chinese

ADOBE – Exhibit 1012, page 16

names. The Pinyin Romanization of Chinese syllables was used, combining syllables together into one-, two-, and three-syllable words. Because no tests were done to determine whether the words actually made sense, an exhaustive search was initiated. Since there are 298 Chinese syllables in the Pinyin system, there are 158,404 two-syllable words, and slightly more than 16,000,000 three-syllable words. A similar mode of attack could as easily be used with English, using rules for building pronounceable nonsense words.

6. Word pairs. The magnitude of an exhaustive test of this nature is staggering. To simplify the test, only words of three or four characters in length from */usr/dict/words* were used. Even so, the number of word pairs is about ten million.

A dictionary attack is much more powerful when it is used against a file of keys and not a single key. A single user may be smart enough to choose good keys. If a thousand people each choose their own key as a password to a computer system, the odds are excellent that at least one person will choose a key in the attacker's dictionary.

Random Keys

Good keys are random-bit strings generated by some automatic process. If the key is 64 bits long, every possible 64-bit key must be equally likely. Generate the key bits from either a reliably random source (see Section 17.14) or a cryptographically secure pseudo-random-bit generator (see Chapters 16 and 17.) If these automatic processes are unavailable, flip a coin or roll a die.

This is important, but don't get too caught up in arguing about whether random noise from audio sources is more random than random noise from radioactive decay. None of these random-noise sources will be perfect, but they will probably be good enough. It is important to use a good random-number generator for key generation, but it is far more important to use good encryption algorithms and key management procedures. If you are worried about the randomness of your keys, use the key-crunching technique described below.

Some encryption algorithms have weak keys: specific keys that are less secure than the other keys. I advise testing for these weak keys and generating a new one if you discover one. DES has only 16 weak keys out of 2^{56}, so the odds of generating any of these keys are incredibly small. It has been argued that a cryptanalyst would have no idea that a weak key is being used and therefore gains no advantage from their accidental use. It has also been argued that not using weak keys gives a cryptanalyst information. However, testing for the few weak keys is so easy that it seems imprudent not to do so.

Generating keys for public-key cryptography systems is harder, because often the keys must have certain mathematical properties (they may have to be prime, be a quadratic residue, etc.). Techniques for generating large random prime numbers are discussed in Section 11.5. The important thing to remember from a key management point of view is that the random seeds for those generators must be just that: random.

ADOBE – Exhibit 1012, page 17

Generating a random key isn't always possible. Sometimes you need to remember your key. (See how long it takes you to remember 25e8 56f2 e8ba c820). If you have to generate an easy-to-remember key, make it obscure. The ideal would be something easy to remember, but difficult to guess. Here are some suggestions:

— Word pairs separated by a punctuation character, for example "turtle*moose" or "zorch!splat"

— Strings of letters that are an acronym of a longer phrase; for example, "Mein Luftkissenfahrzeug ist voller Aale!" generates the key "MLivA!"

Pass Phrases

A better solution is to use an entire phrase instead of a word, and to convert that phrase into a key. These phrases are called **pass phrases**. A technique called **key crunching** converts the easy-to-remember phrases into random keys. Use a one-way hash function to transform an arbitrary-length text string into a pseudo-random-bit string.

For example, the easy-to-remember text string:

```
My name is Ozymandias, king of kings. Look on my works, ye mighty, and despair.
```

might crunch into this 64-bit key:

```
e6c1 4398 5ae9 0a9b
```

Of course, it can be difficult to type an entire phrase into a computer with the echo turned off. Clever suggestions to solve this problem would be appreciated.

If the phrase is long enough, the resulting key will be random. Exactly what "long enough" means is open to interpretation. Information theory tells us that standard English has about 1.3 bits of information per character (see Section 11.1). For a 64-bit key, a pass phrase of about 49 characters, or 10 normal English words, should be sufficient. As a rule of thumb, figure that you need five words for each 4 bytes of key. That's a conservative assumption, since it doesn't take into account case, spacing, and punctuation.

This technique can even be used to generate private keys for public-key cryptography systems: The text string could be crunched into a random seed, and that seed could be fed into a deterministic system that generates public-key/private-key key pairs.

If you are choosing a pass phrase, choose something unique and easy-to-remember. Don't choose phrases from literature—the example from "Ozymandias" is a bad one. Both the complete works of Shakespeare and the dialogue from *Star Wars* are available on-line and can be used in a dictionary attack. Choose something obscure, but personal. Include punctuation and capitalization; if you can, include numbers and non-alphanumeric symbols. Poor or improper English, or even a foreign language, makes the pass phrase less susceptible to a dictionary attack. One suggestion is to use a phrase that is "shocking nonsense": something offensive enough that you are likely to remember and unlikely to write down.

ADOBE – Exhibit 1012, page 18

Despite everything written here, obscurity is no substitute for true randomness. The best keys are random keys, difficult as they are to remember.

X9.17 Key Generation

The ANSI X9.17 standard specifies a method of key generation (see Figure 8.1) [55]. This does not generate easy-to-remember keys; it is more suitable for generating session keys or pseudo-random numbers within a system. The cryptographic algorithm used to generate keys is DES, but it could just as easily be any algorithm.

Let $E_K[X]$ be DES encryption of X with key K. This is a special key reserved for secret key generation. V_0 is a secret 64-bit seed. T is a timestamp. To generate the random key R_i, calculate:

$$R_i = E_K[E_K[T_i] \oplus V_i]$$

To generate V_{i+1}, calculate:

$$V_{i+1} = E_K[E_K[T_i] \oplus R_i]$$

To turn R_i into a DES key, simply adjust every eighth bit for parity. If you need a 64-bit key, use it as is. If you need a 128-bit key, generate a pair of keys and concatenate them together.

DoD Key Generation

The U.S. Department of Defense recommends using DES in OFB mode (see Section 9.8) to generate random keys [1144]. Generate a DES key from system interrupt vectors, system status registers, and system counters. Generate an initialization vector from the system clock, system ID, and date and time. For the plaintext, use an externally generated 64-bit quantity: eight characters typed in by a system administrator, for example. Use the output as your key.

8.2 Nonlinear Keyspaces

Imagine that you are a military cryptography organization, building a piece of cryptography equipment for your troops. You want to use a secure algorithm, but you are

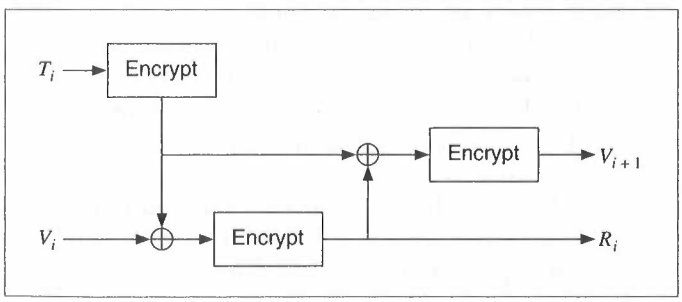


Figure 8.1 ANSI X9.17 key generation.

ADOBE – Exhibit 1012, page 19

worried about the equipment falling into enemy hands. The last thing you want is for your enemy to be able to use the equipment to protect *their* secrets.

If you can put your algorithm in a tamperproof module, here's what you can do. You can require keys of a special and secret form; all other keys will cause the module to encrypt and decrypt using a severely weakened algorithm. You can make it so that the odds of someone, not knowing this special form but accidentally stumbling on a correct key, are vanishingly small.

This is called a **nonlinear keyspace**, because all the keys are not equally strong. (The opposite is a linear, or **flat**, keyspace.) An easy way to do this is to create the key as two parts: the key itself and some fixed string encrypted with that key. The module decrypts the string with the key; if it gets the fixed string it uses the key normally, if not it uses a different, weak algorithm. If the algorithm has a 128-bit key and a 64-bit block size, the overall key is 192 bits; this gives the algorithm an effective key of 2^{128}, but makes the odds of randomly choosing a good key one in 2^{64}.

You can be even subtler. You can design an algorithm such that certain keys are stronger than others. An algorithm can have no weak keys—keys that are obviously very poor—and can still have a nonlinear keyspace.

This only works if the algorithm is secret and the enemy can't reverse-engineer it, or if the difference in key strength is subtle enough that the enemy can't figure it out. The NSA did this with the secret algorithms in their Overtake modules (see Section 25.1). Did they do the same thing with Skipjack (see Section 13.12)? No one knows.

8.3 TRANSFERRING KEYS

Alice and Bob are going to use a symmetric cryptographic algorithm to communicate securely; they need the same key. Alice generates a key using a random-key generator. Now she has to give it to Bob—securely. If Alice can meet Bob somewhere (a back alley, a windowless room, or one of Jupiter's moons), she can give him a copy of the key. Otherwise, they have a problem. Public-key cryptography solves the problem nicely and with a minimum of prearrangement, but these techniques are not always available (see Section 3.1). Some systems use alternate channels known to be secure. Alice could send Bob the key with a trusted messenger. She could send it by certified mail or via an overnight delivery service. She could set up another communications channel with Bob and hope no one is eavesdropping on that one.

Alice could send Bob the symmetric key over their communications channel—the one they are going to encrypt. This is foolish; if the channel warrants encryption, sending the encryption key in the clear over the same channel guarantees that anyone eavesdropping on the channel can decrypt all communications.

The X9.17 standard [55] specifies two types of keys: key-encryption keys and data keys. **Key-Encryption Keys** encrypt other keys for distribution. **Data Keys** encrypt message traffic. These key-encrypting keys have to be distributed manually (although they can be secured in a tamperproof device, like a smart card), but only seldomly. Data keys are distributed more often. More details are in [75]. This two-tiered key concept is used a lot in key distribution.

ADOBE – Exhibit 1012, page 20

Another solution to the distribution problem splits the key into several different parts (see Section 3.6) and sends each of those parts over a different channel. One part could be sent over the telephone, one by mail, one by overnight delivery service, one by carrier pigeon, and so on. (see Figure 8.2). Since an adversary could collect all but one of the parts and still have no idea what the key is, this method will work in all but extreme cases. Section 3.6 discusses schemes for splitting a key into several parts. Alice could even use a secret sharing scheme (see Section 3.7), allowing Bob to reconstruct the key if some of the shares are lost in transmission.

Alice sends Bob the key-encryption key securely, either by a face-to-face meeting or the splitting technique just discussed. Once Alice and Bob both have the key-encryption key, Alice can send Bob daily data keys over the same communications channel. Alice encrypts each data key with the key-encryption key. Since the amount of traffic being encrypted with the key-encryption key is low, it does not have to be changed as often. However, since compromise of the key-encryption key could compromise every message encrypted with every key that was encrypted with the key-encryption key, it must be stored securely.

Key Distribution in Large Networks

Key-encryption keys shared by pairs of users work well in small networks, but can quickly get cumbersome if the networks become large. Since every pair of users must exchange keys, the total number of key exchanges required in an n-person network is $n(n-1)/2$.

In a six-person network, 15 key exchanges are required. In a 1000-person network, nearly 500,000 key exchanges are required. In these cases, creating a central key server (or servers) makes the operation much more efficient.

Alternatively, any of the symmetric-cryptography or public-key-cryptography protocols in Section 3.1 provides for secure key distribution.

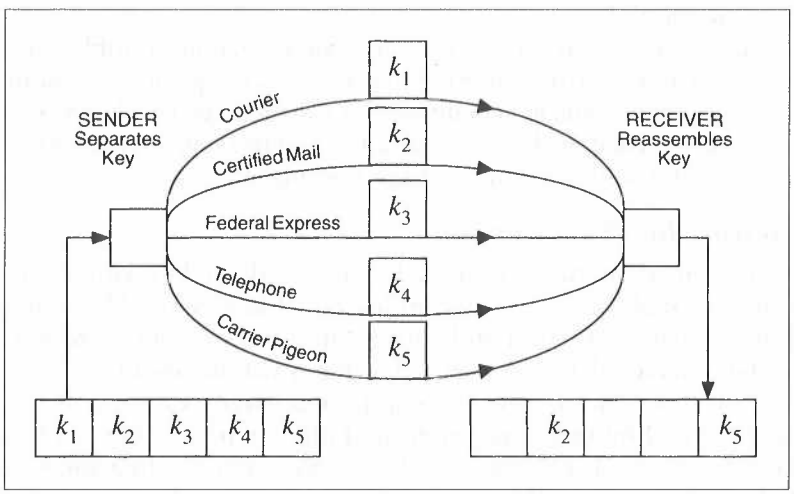


Figure 8.2 Key distribution via parallel channels.

ADOBE – Exhibit 1012, page 21

8.4 VERIFYING KEYS

When Bob receives a key, how does he know it came from Alice and not from someone pretending to be Alice? If Alice gives it to him when they are face-to-face, it's easy. If Alice sends her key via a trusted courier, then Bob has to trust the courier. If the key is encrypted with a key-encryption key, then Bob has to trust the fact that only Alice has that key. If Alice uses a digital signature protocol to sign the key, Bob has to trust the public-key database when he verifies that signature. (He also has to trust that Alice has kept her key secure.) If a Key Distribution Center (KDC) signs Alice's public key, Bob has to trust that his copy of the KDC's public key has not been tampered with.

In the end, someone who controls the entire network around Bob can make him think whatever he likes. Mallory could send an encrypted and signed message purporting to be from Alice. When Bob tried to access the public-key database to verify Alice's signature, Mallory could substitute his own public key. Mallory could invent his own false KDC and exchange the real KDC's public key for his own creation. Bob wouldn't be the wiser.

Some people have used this argument to claim that public-key cryptography is useless. Since the only way for Alice and Bob to ensure that their keys have not been tampered with is to meet face-to-face, public-key cryptography doesn't enhance security at all.

This view is naïve. It is theoretically true, but reality is far more complicated. Public-key cryptography, used with digital signatures and trusted KDCs, makes it much more difficult to substitute one key for another. Bob can never be absolutely certain that Mallory isn't controlling his entire reality, but Bob can be confident that doing so requires more resources than most real-world Mallorys have access to.

Bob could also verify Alice's key over the telephone, where he can hear her voice. Voice recognition is a really good authentication scheme. If it's a public key, he can safely recite it in public. If it's a secret key, he can use a one-way hash function to verify the key. Both PGP (see Section 24.12) and the AT&T TSD (see Section 24.18) use this kind of key verification.

Sometimes, it may not even be important to verify exactly whom a public key belongs to. It may be necessary to verify that it belongs to the same person to whom it belonged last year. If someone sends a signed withdrawal message to a bank, the bank does not have to be concerned with who withdraws the money, only whether it is the same person who deposited the money in the first place.

Error Detection during Key Transmission

Sometimes keys get garbled in transmission. Since a garbled key can mean megabytes of undecryptable ciphertext, this is a problem. All keys should be transmitted with some kind of error detection and correction bits. This way errors in transmission can be easily detected and, if required, the key can be resent.

One of the most widely used methods is to encrypt a constant value with the key, and to send the first 2 to 4 bytes of that ciphertext along with the key. At the receiving end, do the same thing. If the encrypted constants match, then the key has been transmitted without error. The chance of an undetected error ranges from one in 2^{16} to one in 2^{32}.

ADOBE – Exhibit 1012, page 22

Key-error Detection during Decryption

Sometimes the receiver wants to check if a particular key he has is the correct symmetric decryption key. If the plaintext message is something like ASCII, he can try to decrypt and read the message. If the plaintext is random, there are other tricks.

The naïve approach is to attach a **verification block**: a known header to the plaintext message before encryption. At the receiving end, Bob decrypts the header and verifies that it is correct. This works, but it gives Eve a known plaintext to help cryptanalyze the system. It also makes attacks against short-key ciphers like DES and all exportable ciphers easy. Precalculate the checksum once for each key, then use that checksum to determine the key in any message you intercept after that. This is a feature of *any* key checksum that doesn't include random or at least different data in each checksum. It's very similar in concept to using salt when generating keys from passphrases.

Here's a better way to do this [821]:

(1) Generate an IV (not the one used for the message).
(2) Use that IV to generate a large block of bits: say, 512.
(3) Hash the result.
(4) Use the same fixed bits of the hash, say 32, for the key checksum.

This gives Eve some information, but very little. If she tries to use the low 32 bits of the final hash value to mount a brute-force attack, she has to do multiple encryptions plus a hash per candidate key; brute-force on the key itself would be quicker.

She also gets no known-plaintext values to try out, and even if she manages to choose our random value for us, she never gets a chosen-plaintext out of us, since it goes through the hash function before she sees it.

8.5 Using Keys

Software encryption is scary. Gone are the days of simple microcomputers under the control of single programs. Now there's Macintosh System 7, Windows NT, and UNIX. You can't tell when the operating system will suspend the encryption application in progress, write everything to disk, and take care of some pressing task. When the operating system finally gets back to encrypting whatever is being encrypted, everything will look just fine. No one will ever realize that the operating system wrote the encryption application to disk, and that it wrote the key along with it. The key will sit on the disk, unencrypted, until the computer writes over that area of memory again. It could be minutes or it could be months. It could even be never; the key could still be sitting there when an adversary goes over the hard drive with a fine-tooth comb. In a preemptive, multitasking environment, you can set your encryption operation to a high enough priority so it will not be interrupted. This would mitigate the risk. Even so, the whole thing is dicey at best.

Hardware implementations are safer. Many encryption devices are designed to erase the key if tampered with. For example, the IBM PS/2 encryption card has an

ADOBE – Exhibit 1012, page 23

epoxy unit containing the DES chip, battery, and memory. Of course, you have to trust the hardware manufacturer to implement the feature properly.

Some communications applications, such as telephone encryptors, can use **session keys**. A session key is a key that is just used for one communications session—a single telephone conversation—and then discarded. There is no reason to store the key after it has been used. And if you use some key-exchange protocol to transfer the key from one conversant to the other, the key doesn't have to be stored before it is used either. This makes it far less likely that the key might be compromised.

Controlling Key Usage

In some applications it may be desirable to control how a session key is used. Some users may need session keys only for encryption or only for decryption. Session keys might only be authorized for use on a certain machine or at a certain time. One scheme to handle these sorts of restrictions attaches a **Control Vector** (CV) to the key; the control vector specifies the uses and restrictions for that key (see Section 24.1) [1025,1026]. This CV is hashed and XORed with a master key; the result is used as an encryption key to encrypt the session key. The resultant encrypted session key is then stored with the CV. To recover the session key, hash the CV and XOR it with the master key, and use the result to decrypt the encrypted session key.

The advantages of this scheme are that the CV can be of arbitrary length and that it is always stored in the clear with the encrypted key. This scheme assumes quite a bit about tamperproof hardware and the inability of users to get at the keys directly. This system is discussed further in Sections 24.1 and 24.8.

8.6 UPDATING KEYS

Imagine an encrypted data link where you want to change keys daily. Sometimes it's a pain to distribute a new key every day. An easier solution is to generate a new key from the old key; this is sometimes called **key updating**.

All it takes is a one-way function. If Alice and Bob share the same key and they both operate on it using the same one-way function, they will get the same result. Then they can take the bits they need from the results to create the new key.

Key updating works, but remember that the new key is only as secure as the old key was. If Eve managed to get her hands on the old key, she can perform the key updating function herself. However, if Eve doesn't have the old key and is trying a ciphertext-only attack on the encrypted traffic, this is a good way for Alice and Bob to protect themselves.

8.7 STORING KEYS

The least complex key storage problem is that of a single user, Alice, encrypting files for later use. Since she is the only person involved, she is the only person responsible for the key. Some systems take the easy approach: The key is stored in Alice's brain and never on the system. Alice is responsible for remembering the key and entering it every time she needs a file encrypted or decrypted.

ADOBE – Exhibit 1012, page 24

An example of this system is IPS [881]. Users can either directly enter the 64-bit key or enter the key as a longer character string. The system then generates a 64-bit key from the character string using a key-crunching technique.

Another solution is to store the key in a magnetic stripe card, plastic key with an embedded ROM chip (called a **ROM key**), or smart card [556,557,455]. A user could then enter his key into the system by inserting the physical token into a special reader in his encryption box or attached to his computer terminal. While the user can use the key, he does not know it and cannot compromise it. He can use it only in the way and for the purposes indicated by the control vector.

A ROM key is a very clever idea. People understand physical keys, what they signify and how to protect them. Putting a cryptographic key in the same physical form makes storing and protecting that key more intuitive.

This technique is made more secure by splitting the key into two halves, storing one half in the terminal and the other half in the ROM key. The U.S. government's STU-III secure telephone works this way. Losing the ROM key does not compromise the cryptographic key—change that key and everything is back to normal. The same is true with the loss of the terminal. This way, compromising either the ROM key or the system does not compromise the cryptographic key—an adversary must have both parts.

Hard-to-remember keys can be stored in encrypted form, using something similar to a key-encryption key. For example, an RSA private key could be encrypted with a DES key and stored on disk. To recover the RSA key, the user has to type in the DES key to a decryption program.

If the keys are generated deterministically (with a cryptographically secure pseudo-random-sequence generator), it might be easier to regenerate the keys from an easy-to-remember password every time they are required.

Ideally, a key should never appear unencrypted outside the encryption device. This isn't always possible, but it is a worthy goal.

8.8 Backup Keys

Alice is the chief financial officer at Secrets, Ltd.—"We don't tell you our motto." Like any good corporate officer, she follows the company's security guidelines and encrypts all her data. Unfortunately, she ignores the company's street-crossing guidelines and gets hit by a truck. What does the company's president, Bob, do?

Unless Alice left a copy of her key, he's in deep trouble. The whole point of encryption is to make files unrecoverable without the key. Unless Alice was a moron and used lousy encryption software, her files are gone forever.

Bob can avoid this in several ways. The simplest is sometimes called **key escrow** (see Section 4.14): He requires all employees to write their keys on paper and give them to the company's security officer, who will lock them in a safe somewhere (or encrypt them all with a master key). Now, when Alice is bowled over on the Interstate, Bob can ask his security officer for her key. Bob should make sure to have the combination to the safe himself as well; otherwise, if the security officer is run over by another truck, Bob will be out of luck again.

ADOBE – Exhibit 1012, page 25

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.

ADOBE – Exhibit 1012, page 26

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?

ADOBE – Exhibit 1012, page 27

— 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].

ADOBE – Exhibit 1012, page 28

Keys must be destroyed securely (see Section 10.9). If the key is written on paper, the paper should be shredded or burned. Be careful to use a high-quality shredder; many lousy shredders are on the market. Algorithms in this book are secure against brute-force attacks costing millions of dollars and taking millions of years. If an adversary can recover your key by taking a bag of shredded documents from your trash and paying 100 unemployed workers in some backwater country ten cents per hour for a year to piece the shredded pages together, that would be $26,000 well spent.

If the key is in a hardware EEPROM, the key should be overwritten multiple times. If the key is in a hardware EPROM or PROM, the chip should be smashed into tiny bits and scattered to the four winds. If the key is stored on a computer disk, the actual bits of the storage should be overwritten multiple times (see Section 10.9) or the disk should be shredded.

A potential problem is that, in a computer, keys can be easily copied and stored in multiple locations. Any computer that does its own memory management, constantly swapping programs in and out of memory, exacerbates the problem. There is no way to ensure that successful key erasure has taken place in the computer, especially if the computer's operating system controls the erasure process. The more paranoid among you should consider writing a special erasure program that scans all disks looking for copies of the key's bit pattern on unused blocks and then erases those blocks. Also remember to erase the contents of any temporary, or "swap," files.

8.12 Public-Key Key Management

Public-key cryptography makes key management easier, but it has its own unique problems. Each person has only one public key, regardless of the number of people on the network. If Alice wants to send a message to Bob, she has to get Bob's public key. She can go about this several ways:

- She can get it from Bob.
- She can get it from a centralized database.
- She can get it from her own private database.

Section 2.5 discussed a number of possible attacks against public-key cryptography, based on Mallory substituting his key for Bob's. The scenario is that Alice wants to send a message to Bob. She goes to the public-key database and gets Bob's public key. But Mallory, who is sneaky, has substituted his own key for Bob's. (If Alice asks Bob directly, Mallory has to intercept Bob's transmission and substitute his key for Bob's.) Alice encrypts her message in Mallory's key and sends it to Bob. Mallory intercepts the message, decrypts it, and reads it. He re-encrypts it with Bob's real key and sends it on to Bob. Neither Alice nor Bob is the wiser.

Public-key Certificates

A **public-key certificate** is someone's public key, signed by a trustworthy person. Certificates are used to thwart attempts to substitute one key for another [879]. Bob's

ADOBE – Exhibit 1012, page 29

certificate, in the public-key database, contains a lot more than his public key. It contains information about Bob—his name, address, and so on—and it is signed by someone Alice trusts: Trent (usually known as a **certification authority**, or CA). By signing both the key and the information about Bob, Trent certifies that the information about Bob is correct and that the public key belongs to Bob. Alice checks Trent's signature and then uses the public key, secure in the knowledge that it is Bob's and no one else's. Certificates play an important role in a number of public-key protocols such as PEM [825] (see Section 24.10) and X.509 [304] (see Section 24.9).

A complicated noncryptographic issue surrounds this type of system. What is the meaning of certification? Or, to put it another way, who is trusted to issue certificates to whom? Anyone may sign anyone else's certificate, but there needs to be some way to filter out questionable certificates: for example, certificates for employees of one company signed by the CA of another company. Normally, a certification chain transfers trust: A single trusted entity certifies trusted agents, trusted agents certify company CAs, and company CAs certify their employees.

Here are some more things to think about:

- What level of trust in someone's identity is implied by his certificate?
- What are the relationships between a person and the CA that certified his public key, and how can those relationships be implied by the certificate?
- Who can be trusted to be the "single trusted entity" at the top of the certification chain?
- How long can a certification chain be?

Ideally, Bob would follow some kind of authentication procedure before the CA signs his certificate. Additionally, some kind of timestamp or an indication of the certificate's validity period is important to guard against compromised keys [461].

Timestamping is not enough. Keys may be invalidated before they have expired, either through compromise or for administrative reasons. Hence, it is important the CA keep a list of invalid certificates, and for users to regularly check that list. This key revocation problem is still a difficult one to solve.

And one public-key/private-key pair is not enough. Certainly any good implementation of public-key cryptography needs separate keys for encryption and digital signatures. This separation allows for different security levels, expiration times, backup procedures, and so on. Someone might sign messages with a 2048-bit key stored on a smart card and good for twenty years, while they might use a 768-bit key stored in the computer and good for six months for encryption.

And a single pair of encryption and signature keys isn't enough, either. A private key authenticates a relationship as well as an identity, and people have more than one relationship. Alice might want to sign one document as Alice the individual, another as Alice, vice-president of Monolith, Inc., and a third as Alice, president of her community organization. Some of these keys are more valuable than others, so they can be better protected. Alice might have to store a backup of her work key

ADOBE – Exhibit 1012, page 30

with the company's security officer; she doesn't want the company to have a copy of the key she signed her mortgage with. Just as Alice has multiple physical keys in her pocket, she is going to have multiple cryptographic keys.

Distributed Key Management

In some situations, this sort of centralized key management will not work. Perhaps there is no CA whom Alice and Bob both trust. Perhaps Alice and Bob trust only their friends. Perhaps Alice and Bob trust no one.

Distributed key management, used in PGP (see Section 24.12), solves this problem with **introducers**. Introducers are other users of the system who sign their friends' public keys. For example, when Bob generates his public key, he gives copies to his friends: Carol and Dave. They know Bob, so they each sign Bob's key and give Bob a copy of the signature. Now, when Bob presents his key to a stranger, Alice, he presents it with the signatures of these two introducers. If Alice also knows and trusts Carol, she has reason to believe that Bob's key is valid. If she knows and trusts Carol and Dave a little, she has reason to believe that Bob's key is valid. If she doesn't know either Carol or Dave, she has no reason to trust Bob's key.

Over time, Bob will collect many more introducers. If Alice and Bob travel in similar circles, the odds are good that Alice will know one of Bob's introducers. To prevent against Mallory's substituting one key for another, an introducer must be sure that Bob's key belongs to Bob before he signs it. Perhaps the introducer should require the key be given face-to-face or verified over the telephone.

The benefit of this mechanism is that there is no CA that everyone has to trust. The down side is that when Alice receives Bob's public key, she has no guarantee that she will know any of the introducers and therefore no guarantee that she will trust the validity of the key.

ADOBE – Exhibit 1012, page 31

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 has 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 $H(M) = H(M')$.

ADOBE – Exhibit 1012, page 32

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.

ADOBE – Exhibit 1012, page 33

(2) Append 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.

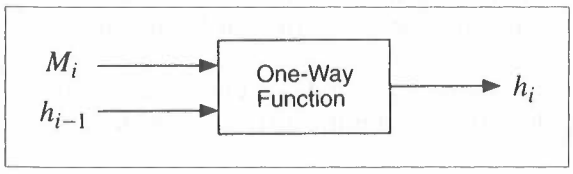


Figure 18.1 One-way function.

ADOBE – Exhibit 1012, page 34

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.

ADOBE – Exhibit 1012, page 35

18.3 *N*-HASH

N-Hash is an algorithm invented by researchers at Nippon Telephone and Telegraph, the same people who invented FEAL, in 1990 [1105,1106]. *N*-Hash uses 128-bit message blocks, a complicated randomizing function similar to FEAL's, and produces a 128-bit hash value.

The hash of each 128-bit block is a function of the block and the hash of the previous block.

$$H_0 = I, \text{ where } I \text{ is a random initial value}$$
$$H_i = g(M_i, H_{i-1}) \oplus M_i \oplus H_{i-1}$$

The hash of the entire message is the hash of the last message block. The random initial value, I, can be any value determined by the user (even all zeros).

The function g is a complicated one. Figure 18.2 is an overview of the algorithm. Initially, the 128-bit hash of the previous message block, H_{i-1}, has its 64-bit left half

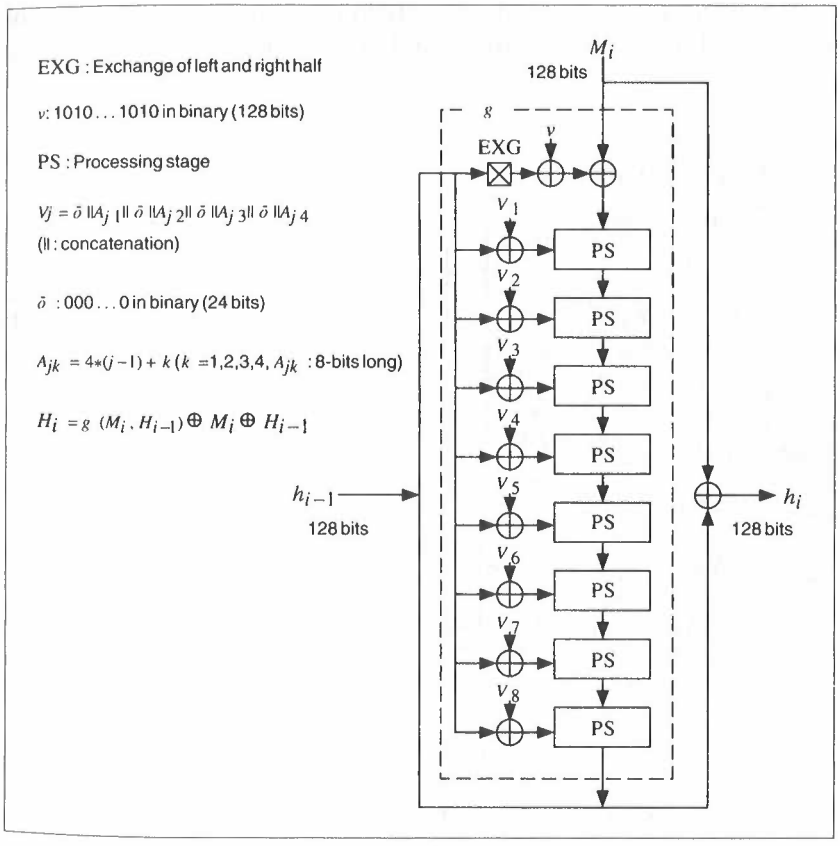


Figure 18.2 Outline of N-*Hash.*

ADOBE – Exhibit 1012, page 36

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

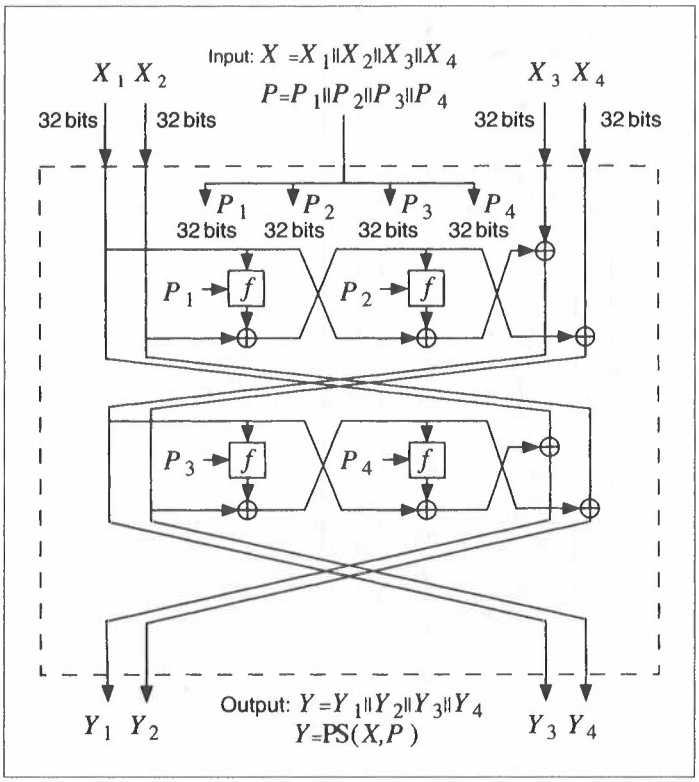


Figure 18.3 One processing stage of N-*Hash.*

ADOBE – Exhibit 1012, page 37

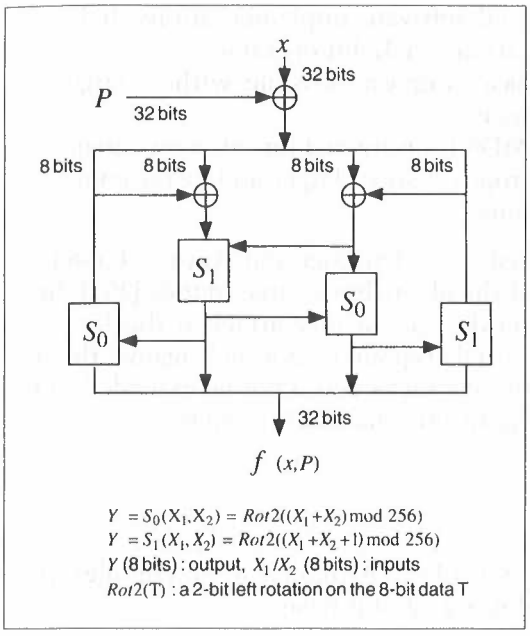


Figure 18.4 Function f.

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.

ADOBE – Exhibit 1012, page 38

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-

ADOBE – Exhibit 1012, page 39

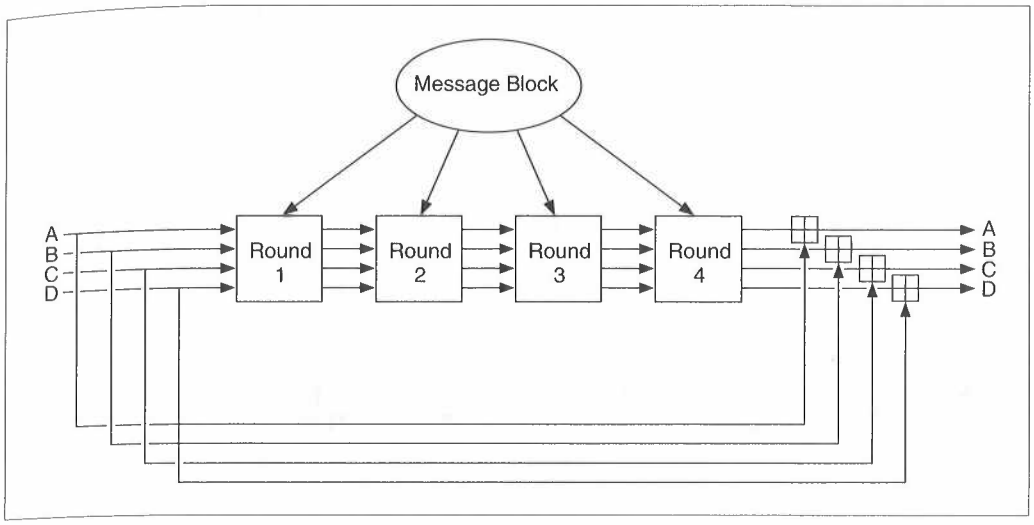


Figure 18.5 MD5 main loop.

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 jth sub-block of the message (from 0 to 15), and $<<<s$ represents a left circular shift of s bits, the four operations are:

$$FF(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + F(b,c,d) + M_j + t_i) <<< s)$$
$$GG(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + G(b,c,d) + M_j + t_i) <<< s)$$
$$HH(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + H(b,c,d) + M_j + t_i) <<< s)$$
$$II(a,b,c,d,M_j,s,t_i) \text{ denotes } a = b + ((a + I(b,c,d) + M_j + t_i) <<< s)$$

ADOBE – Exhibit 1012, page 40

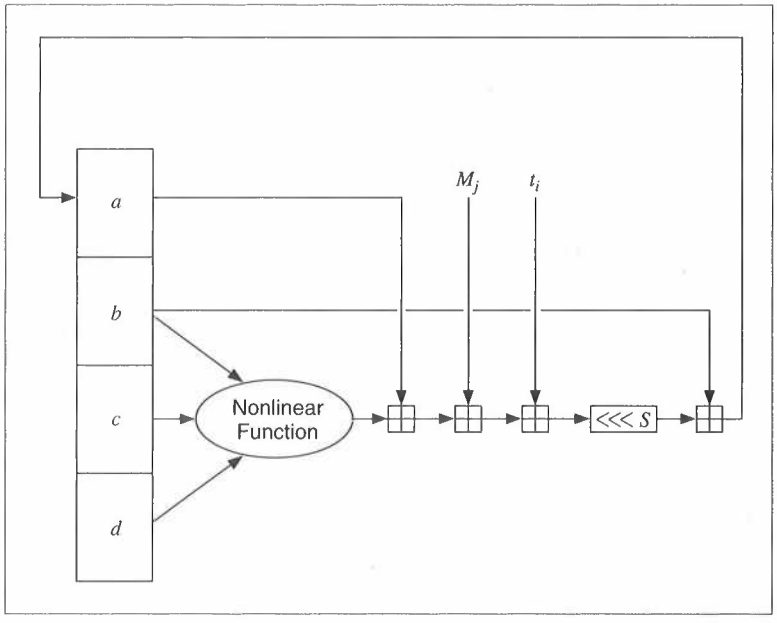


Figure 18.6 One MD5 operation.

The four rounds (64 steps) look like:

Round 1:

FF (*a*, *b*, *c*, *d*, M_0, 7, 0xd76aa478)
FF (*d*, *a*, *b*, *c*, M_1, 12, 0xe8c7b756)
FF (*c*, *d*, *a*, *b*, M_2, 17, 0x242070db)
FF (*b*, *c*, *d*, *a*, M_3, 22, 0xc1bdceee)
FF (*a*, *b*, *c*, *d*, M_4, 7, 0xf57c0faf)
FF (*d*, *a*, *b*, *c*, M_5, 12, 0x4787c62a)
FF (*c*, *d*, *a*, *b*, M_6, 17, 0xa8304613)
FF (*b*, *c*, *d*, *a*, M_7, 22, 0xfd469501)
FF (*a*, *b*, *c*, *d*, M_8, 7, 0x698098d8)
FF (*d*, *a*, *b*, *c*, M_9, 12, 0x8b44f7af)
FF (*c*, *d*, *a*, *b*, M_{10}, 17, 0xffff5bb1)
FF (*b*, *c*, *d*, *a*, M_{11}, 22, 0x895cd7be)
FF (*a*, *b*, *c*, *d*, M_{12}, 7, 0x6b901122)
FF (*d*, *a*, *b*, *c*, M_{13}, 12, 0xfd987193)
FF (*c*, *d*, *a*, *b*, M_{14}, 17, 0xa679438e)
FF (*b*, *c*, *d*, *a*, M_{15}, 22, 0x49b40821)

ADOBE – Exhibit 1012, page 41

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, 0xfffa3942)
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, 0xbebfbc70)
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)

ADOBE – Exhibit 1012, page 42

Round 4:

II (*a*, *b*, *c*, *d*, M_0, 6, 0xf4292244)
II (*d*, *a*, *b*, *c*, M_7, 10, 0x432aff97)
II (*c*, *d*, *a*, *b*, M_{14}, 15, 0xab9423a7)
II (*b*, *c*, *d*, *a*, M_5, 21, 0xfc93a039)
II (*a*, *b*, *c*, *d*, M_{12}, 6, 0x655b59c3)
II (*d*, *a*, *b*, *c*, M_3, 10, 0x8f0ccc92)
II (*c*, *d*, *a*, *b*, M_{10}, 15, 0xffeff47d)
II (*b*, *c*, *d*, *a*, M_1, 21, 0x85845dd1)
II (*a*, *b*, *c*, *d*, M_8, 6, 0x6fa87e4f)
II (*d*, *a*, *b*, *c*, M_{15}, 10, 0xfe2ce6e0)
II (*c*, *d*, *a*, *b*, M_6, 15, 0xa3014314)
II (*b*, *c*, *d*, *a*, M_{13}, 21, 0x4e0811a1)
II (*a*, *b*, *c*, *d*, M_4, 6, 0xf7537e82)
II (*d*, *a*, *b*, *c*, M_{11}, 10, 0xbd3af235)
II (*c*, *d*, *a*, *b*, M_2, 15, 0x2ad7d2bb)
II (*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 $2^{32} * \text{abs}(\sin(i))$, 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.

ADOBE – Exhibit 1012, page 43

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. $S_0, S_1, S_2, \ldots, S_{255}$ 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: $X_0, X_1, X_2, \ldots, X_{47}$. 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
$t = X_k$ 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.

ADOBE – Exhibit 1012, page 44

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

ADOBE – Exhibit 1012, page 45

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:

$$f_t(X,Y,Z) = (X \wedge Y) \vee ((\neg X) \wedge Z), \text{ for } t = 0 \text{ to } 19.$$
$$f_t(X,Y,Z) = X \oplus Y \oplus Z, \text{ for } t = 20 \text{ to } 39.$$
$$f_t(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), \text{ for } t = 40 \text{ to } 59.$$
$$f_t(X,Y,Z) = X \oplus Y \oplus Z, \text{ for } t = 60 \text{ to } 79.$$

Four constants are used in the algorithm:

$$K_t = \text{0x5a827999, for } t = 0 \text{ to } 19.$$
$$K_t = \text{0x6ed9eba1, for } t = 20 \text{ to } 39.$$
$$K_t = \text{0x8f1bbcdc, for } t = 40 \text{ to } 59.$$
$$K_t = \text{0xca62c1d6, for } t = 60 \text{ to } 79.$$

(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:

$$W_t = M_t, \text{ for } t = 0 \text{ to } 15$$
$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) <<< 1, \text{ for } t = 16 \text{ to } 79.$$

(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 tth sub-block of the expanded message, and $<<< s$ represents a left circular shift of s bits, then the main loop looks like:

FOR $t = 0$ to 79
  $TEMP = (a <<< 5) + f_t(b,c,d) + e + W_t + K_t$
  $e = d$
  $d = c$
  $c = b <<< 30$
  $b = a$
  $a = TEMP$

ADOBE – Exhibit 1012, page 46

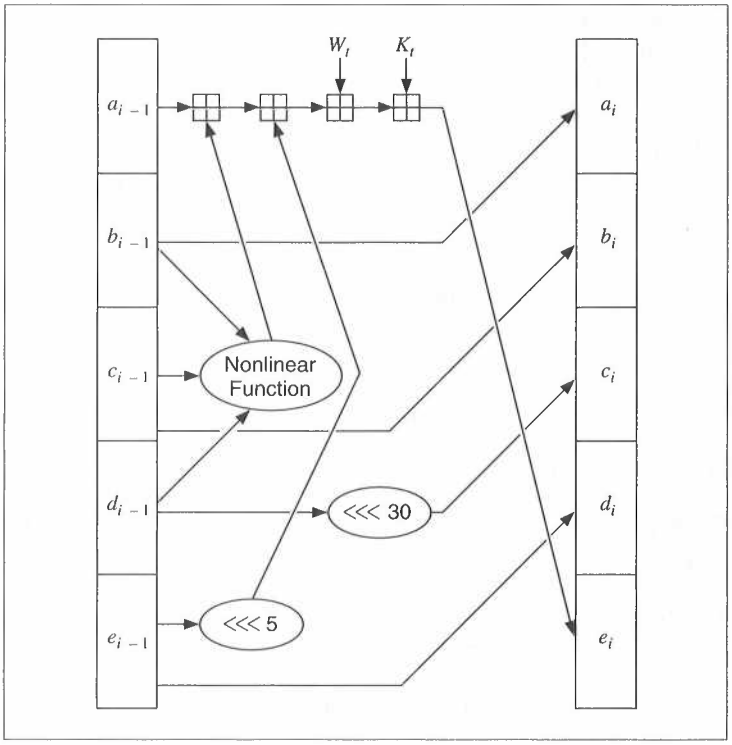


Figure 18.7 One SHA operation.

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 $(XY \vee XZ \vee YZ)$ to $(XZ \vee Y\neg(Z))$ to make *G* less symmetric." SHA uses the MD4 version: $(XY \vee XZ \vee YZ)$.

ADOBE – Exhibit 1012, page 47

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_t. 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 MD5 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.

ADOBE – Exhibit 1012, page 48

The core of the algorithm is

$$TEMP = (f(j,A,B,C,D,E,F,G) <<< 7) + (H <<< 11) + M[i][r(j)] + K(j)$$
$$H = G;\ G = F;\ F = E;\ E = D;\ D = C;\ C = B;\ B = A;\ A = TEMP$$

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. Boognish 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.

ADOBE – Exhibit 1012, page 49

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.

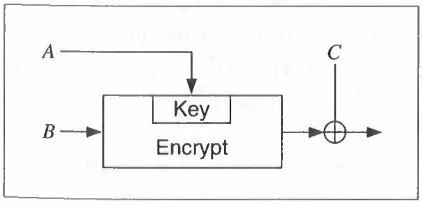


Figure 18.8 General hash function where the hash length equals the block size.

ADOBE – Exhibit 1012, page 50

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.

ADOBE – Exhibit 1012, page 51

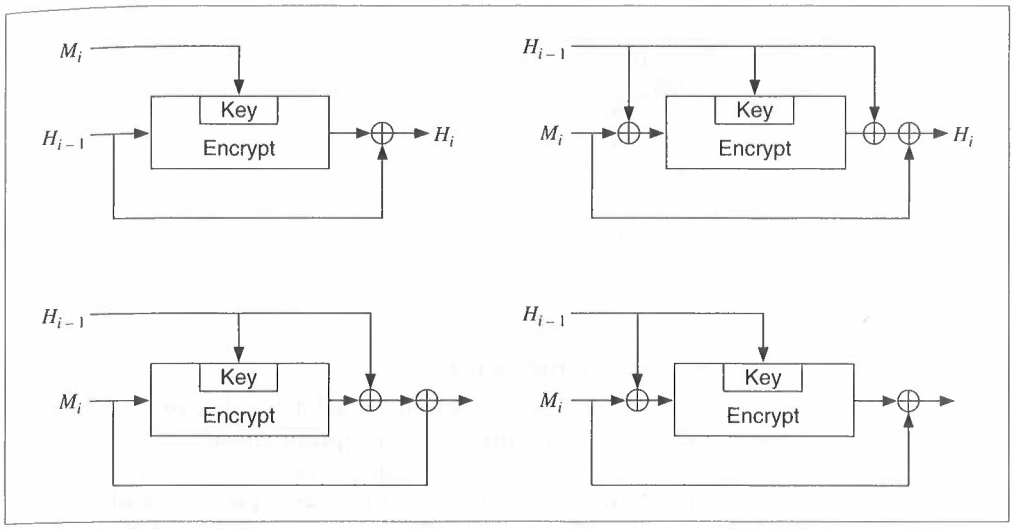


Figure 18.9 The four secure hash functions where the block length equals the hash size.

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.

ADOBE – Exhibit 1012, page 52

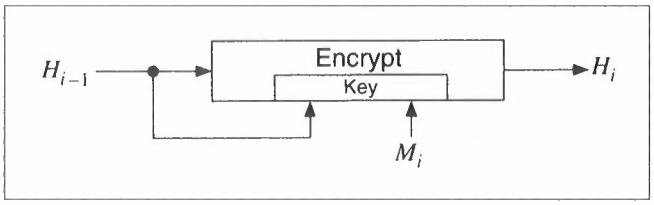


Figure 18.10 Modified Davies-Meyer.

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.

ADOBE – Exhibit 1012, page 53

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 ½ [930,925].

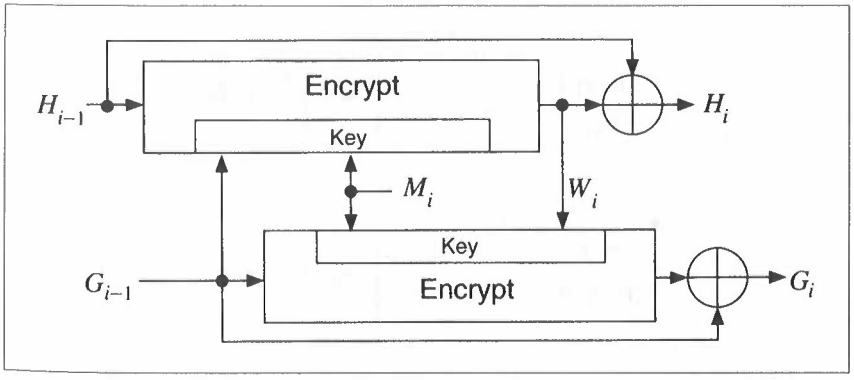


Figure 18.11 Tandem Davies-Meyer.

ADOBE – Exhibit 1012, page 54

In this first scheme, two modified Davies-Meyer functions work in tandem (see Figure 18.11).

$$G_0 = I_G, \text{ where } I_G \text{ is some random initial value}$$
$$H_0 = I_H, \text{ where } I_H \text{ is some other random initial value}$$
$$W_i = E_{G_{i-1}, M_i}(H_{i-1})$$
$$G_i = G_{i-1} \oplus E_{M_i, W_i}(G_{i-1})$$
$$H_i = W_i \oplus H_{i-1}$$

The following scheme uses two modified Davies-Meyer functions side-by-side (see Figure 18.12).

$$G_0 = I_G, \text{ where } I_G \text{ is some random initial value}$$
$$H_0 = I_H, \text{ where } I_H \text{ is some other random initial value}$$
$$G_i = G_{i-1} \oplus E_{M_i, H_{i-1}}(\neg G_{i-1})$$
$$H_i = H_{i-1} \oplus E_{G_{i-1}, M_i}(H_{i-1})$$

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.

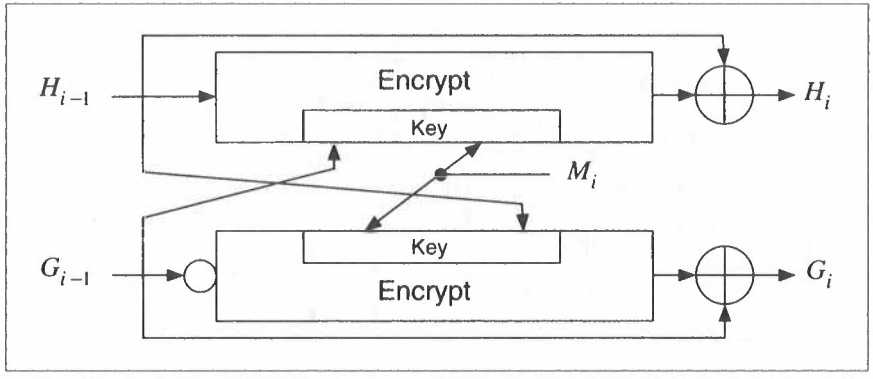


Figure 18.12 Abreast Davies-Meyer.

ADOBE – Exhibit 1012, page 55

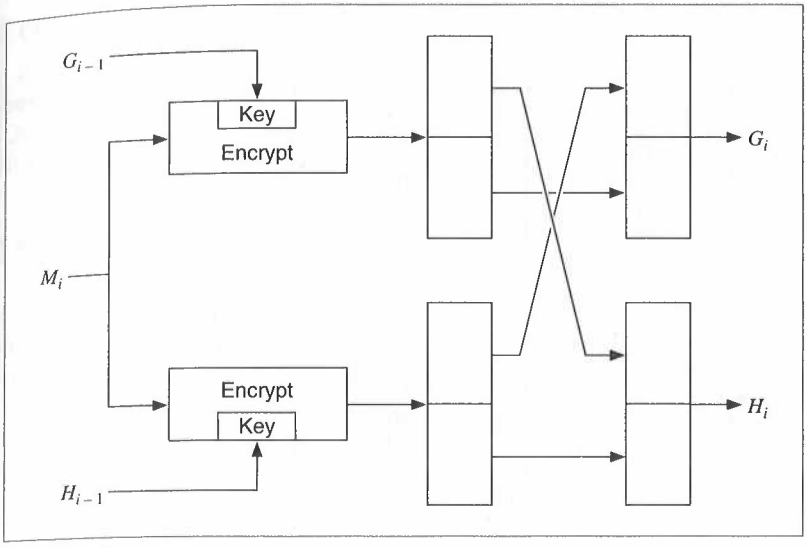


Figure 18.13 MDC-2.

MDC-2 has a hash rate of ½, 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 ¼ (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 ½. The first key is 0x0000000000000000, the second key is 0x2a4152 2f4446502a, 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].

ADOBE – Exhibit 1012, page 56

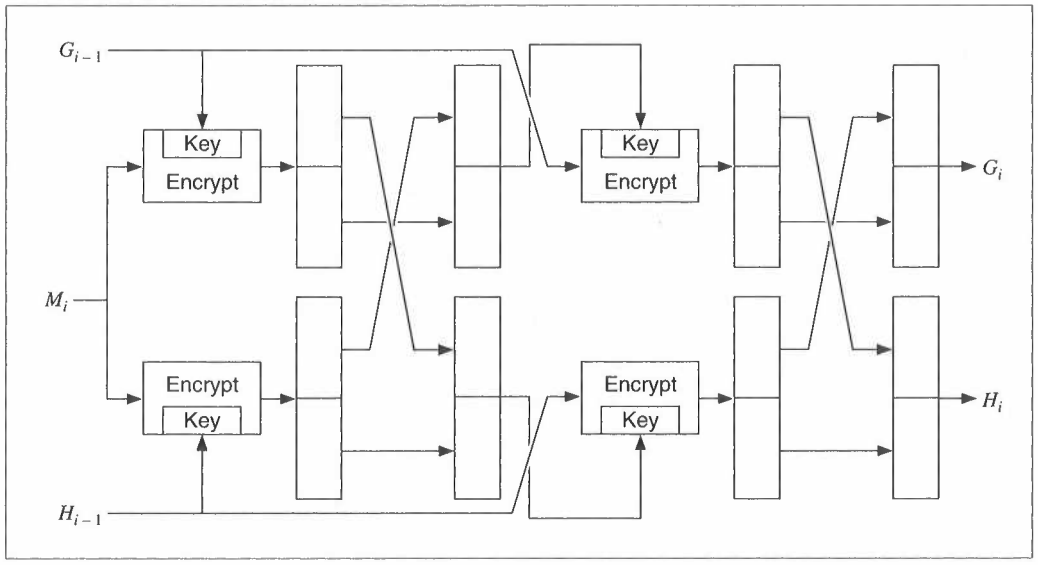


Figure 18.14 MDC-4.

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).

ADOBE – Exhibit 1012, page 57

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

ADOBE – Exhibit 1012, page 58

Table 18.2
Encryption Speeds of Some Hash Functions on a 33 MHz 486SX

Algorithm	Hash Length	Encryption Speed (megabytes/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
SNEERU (4 passes)	128	48
SNEERU (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].

ADOBE – Exhibit 1012, page 59

$$v = v \lll 1$$
$$e = v \oplus w$$
$$x = ((((e + y) \bmod 2^{32}) \vee A \wedge C) * (x \oplus M_i)) \bmod 2^{32} - 1$$
$$y = ((((e + x) \bmod 2^{32}) \vee B \wedge D) * (y \oplus M_i)) \bmod 2^{32} - 2$$

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:

$H_0 = I_H$, where I_H is the secret key
$H_i = (H_{i-1} + M_i)^2 \bmod p$, where p 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

ADOBE – Exhibit 1012, page 60

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:

$H(K_1, H(K_2,M))$

$H(K, H(K,M))$

$H(K,p,M,K)$, where p pads K to a full message block.

ADOBE – Exhibit 1012, page 61

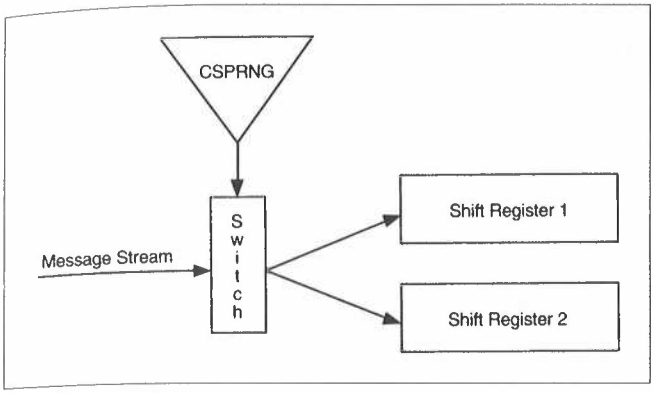


Figure 18.15 Stream cipher MAC.

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.

ADOBE – Exhibit 1012, page 62