APPLIED CRYPTOGRAPHY, SECOND EDITION

PROTOCOLS, ALGORITHMS, AND SOURCE CODE IN C

BRUCE SCHNEIER

20th Anniversary Edition

WILEY

Security First Innovations, LLC, Exhibit 2010
Page 2010 - 1
IPR2024-00212, Google LLC v. Security First Innovations, LLC

Applied Cryptography: Protocols, Algorithms and Source Code in C

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 1996 by Bruce Schneier. All rights reserved.

New foreword copyright © 2015 by Bruce Schneier. All rights reserved.

Published by John Wiley & Sons, Inc.

Published simultaneously in Canada

ISBN: 978-1-119-09672-6

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that Internet websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2015932956

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Security First Innovations, LLC, Exhibit 2010
Page 2010 - 2
IPR2024-00212, Google LLC v. Security First Innovations, LLC

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Contents

Introduction xiii
Foreword by Whitfield Diffie xvii
Preface xxi
How to Read this Book xxii
Acknowledgments xxiv
About the Author xxv

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*

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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*

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

PART II CRYPTOGRAPHIC TECHNIQUES

7 KEY LENGTH *151*
7.1 Symmetric Key Length *151*
7.2 Public-Key Key Length *158*
7.3 Comparing Symmetric and Public-Key Key Length *165*
7.4 Birthday Attacks against One-Way Hash Functions *165*
7.5 How Long Should a Key Be? *166*
7.6 Caveat Emptor *168*

8 KEY MANAGEMENT *169*
8.1 Generating Keys *170*
8.2 Nonlinear Keyspaces *175*
8.3 Transferring Keys *176*
8.4 Verifying Keys *178*
8.5 Using Keys *179*
8.6 Updating Keys *180*
8.7 Storing Keys *180*
8.8 Backup Keys *181*
8.9 Compromised Keys *182*
8.10 Lifetime of Keys *183*
8.11 Destroying Keys *184*
8.12 Public-Key Key Management *185*

9 ALGORITHM TYPES AND MODES *189*
9.1 Electronic Codebook Mode *189*
9.2 Block Replay *191*
9.3 Cipher Block Chaining Mode *193*
9.4 Stream Ciphers *197*
9.5 Self-Synchronizing Stream Ciphers *198*
9.6 Cipher-Feedback Mode *200*
9.7 Synchronous Stream Ciphers *202*
9.8 Output-Feedback Mode *203*
9.9 Counter Mode *205*
9.10 Other Block-Cipher Modes *206*
9.11 Choosing a Cipher Mode *208*
9.12 Interleaving *210*
9.13 Block Ciphers versus Stream Ciphers *210*

10 USING ALGORITHMS *213*
10.1 Choosing an Algorithm *214*
10.2 Public-Key Cryptography versus Symmetric Cryptography *216*
10.3 Encrypting Communications Channels *216*
10.4 Encrypting Data for Storage *220*
10.5 Hardware Encryption versus Software Encryption *223*

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

17.11 Cascading Multiple Stream Ciphers 419
17.12 Choosing a Stream Cipher 420
17.13 Generating Multiple Streams from a Single Pseudo-Random-Sequence Generator 420
17.14 Real Random-Sequence Generators 421

18 ONE-WAY HASH FUNCTIONS 429
18.1 Background 429
18.2 Snefru 431
18.3 *N*-Hash 432
18.4 MD4 435
18.5 MD5 436
18.6 MD2 441
18.7 Secure Hash Algorithm (SHA) 441
18.8 RIPE-MD 445
18.9 HAVAL 445
18.10 Other One-Way Hash Functions 446
18.11 One-Way Hash Functions Using Symmetric Block Algorithms 446
18.12 Using Public-Key Algorithms 455
18.13 Choosing a One-Way Hash Function 455
18.14 Message Authentication Codes 455

19 PUBLIC-KEY ALGORITHMS 461
19.1 Background 461
19.2 Knapsack Algorithms 462
19.3 RSA 466
19.4 Pohlig-Hellman 474
19.5 Rabin 475
19.6 ElGamal 476
19.7 McEliece 479
19.8 Elliptic Curve Cryptosystems 480
19.9 LUC 481
19.10 Finite Automaton Public-Key Cryptosystems 482

20 PUBLIC-KEY DIGITAL SIGNATURE ALGORITHMS 483
20.1 Digital Signature Algorithm (DSA) 483
20.2 DSA Variants 494
20.3 GOST Digital Signature Algorithm 495
20.4 Discrete Logarithm Signature Schemes 496
20.5 Ong-Schnorr-Shamir 498
20.6 ESIGN 499
20.7 Cellular Automata 500
20.8 Other Public-Key Algorithms 500

21 IDENTIFICATION SCHEMES 503
21.1 Feige-Fiat-Shamir 503

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

21.2 Guillou-Quisquater 508
21.3 Schnorr 510
21.4 Converting Identification Schemes to Signature Schemes 512

22 KEY-EXCHANGE ALGORITHMS *513*
22.1 Diffie-Hellman 513
22.2 Station-to-Station Protocol 516
22.3 Shamir's Three-Pass Protocol 516
22.4 COMSET 517
22.5 Encrypted Key Exchange 518
22.6 Fortified Key Negotiation 522
22.7 Conference Key Distribution and Secret Broadcasting 523

23 SPECIAL ALGORITHMS FOR PROTOCOLS *527*
23.1 Multiple-Key Public-Key Cryptography 527
23.2 Secret-Sharing Algorithms 528
23.3 Subliminal Channel 531
23.4 Undeniable Digital Signatures 536
23.5 Designated Confirmer Signatures 539
23.6 Computing with Encrypted Data 540
23.7 Fair Coin Flips 541
23.8 One-Way Accumulators 543
23.9 All-or-Nothing Disclosure of Secrets 543
23.10 Fair and Failsafe Cryptosystems 546
23.11 Zero-Knowledge Proofs of Knowledge 548
23.12 Blind Signatures 549
23.13 Oblivious Transfer 550
23.14 Secure Multiparty Computation 551
23.15 Probabilistic Encryption 552
23.16 Quantum Cryptography 554

PART IV THE REAL WORLD

24 EXAMPLE IMPLEMENTATIONS *561*
24.1 IBM Secret-Key Management Protocol 561
24.2 MITRENET 562
24.3 ISDN 563
24.4 STU-III 565
24.5 Kerberos 566
24.6 KryptoKnight 571
24.7 SESAME 572
24.8 IBM Common Cryptographic Architecture 573
24.9 ISO Authentication Framework 574
24.10 Privacy-Enhanced Mail (PEM) 577
24.11 Message Security Protocol (MSP) 584

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

24.12 Pretty Good Privacy (PGP) *584*
24.13 Smart Cards *587*
24.14 Public-Key Cryptography Standards (PKCS) *588*
24.15 Universal Electronic Payment System (UEPS) *589*
24.16 Clipper *591*
24.17 Capstone *593*
24.18 AT&T Model 3600 Telephone Security Device (TSD) *594*

25 POLITICS *597*
25.1 National Security Agency (NSA) *597*
25.2 National Computer Security Center (NCSC) *599*
25.3 National Institute of Standards and Technology (NIST) *600*
25.4 RSA Data Security, Inc. *603*
25.5 Public Key Partners *604*
25.6 International Association for Cryptographic Research (IACR) *605*
25.7 RACE Integrity Primitives Evaluation (RIPE) *605*
25.8 Conditional Access for Europe (CAFE) *606*
25.9 ISO/IEC 9979 *607*
25.10 Professional, Civil Liberties, and Industry Groups *608*
25.11 Sci.crypt *608*
25.12 Cypherpunks *609*
25.13 Patents *609*
25.14 U.S. Export Rules *610*
25.15 Foreign Import and Export of Cryptography *617*
25.16 Legal Issues *618*

Afterword by Matt Blaze *619*

PART V SOURCE CODE

Source Code *623*

References *675*

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Introduction

I first wrote *Applied Cryptography* in 1993. Two years later, I wrote the greatly expanded second edition. At this vantage point of two decades later, it can be hard to remember how heady cryptography's promise was back then. These were the early days of the Internet. Most of my friends had e-mail, but that was because most of my friends were techies. Few of us used the World Wide Web. There was nothing yet called electronic commerce.

Cryptography was being used by the few who cared. We could encrypt our e-mail with PGP, but mostly we didn't. We could encrypt sensitive files, but mostly we didn't. I don't remember having the option of a usable full-disk encryption product, at least one that I would trust to be reliable.

What we did have were ideas—research and engineering ideas—and that was the point of *Applied Cryptography.* My goal in writing the book was to collect all the good ideas of academic cryptography under one cover and in a form that non-mathematicians could read and use.

What we also had, more important than ideas, was the unshakable belief that technology trumped politics. You can see it in John Perry Barlow's 1996 "Declaration of the Independence of Cyberspace," where he told governments, "You have no moral right to rule us, nor do you possess any methods of enforcement that we have reason to fear." You can see it three years earlier in cypherpunk John Gilmore's famous quote: "The Net interprets censorship as damage and routes around it." You can see it in the pages of *Applied Cryptography.* The first paragraph of the Preface, which I wrote in 1993, says, "There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files. This book is about the latter."

This was the promise of cryptography. It was the promise behind everything—from file and e-mail encryption to digital signatures, digital certified mail, secure election protocols, and digital cash. The math would give us all power and security,

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

because math trumps everything else. It would topple everything from government sovereignty to the music industry's attempts at stopping file sharing.

The "natural law" of cryptography is that it's much easier to use than it is to break. To take a hand-waving example, think about basic encryption. Adding a single bit to a key, say from a 64-bit key to a 65-bit key, adds at most a small amount of work to encrypt and decrypt. But it doubles the amount of work to break. Or, more mathematically, encryption and decryption work grows linearly with key length, but cryptanalysis work grows exponentially. It's always easier for the communicators than the eavesdropper.

It turned out that this was all true, but less important than we had believed. A few years later, we realized that cryptography was just math, and that math has no agency. In order for cryptography to actually do anything, it has to be embedded in a protocol, written in a programming language, embedded in software, run on an operating system and computer attached to a network, and used by living people. All of those things add vulnerabilities and—more importantly—they're more conventionally balanced. That is, there's no inherent advantage for the defender over the attacker. Spending more effort on either results in linear improvements. Even worse, the attacker generally has an inherent advantage over the defender, at least today.

So when we learn about the NSA through the documents provided by Edward Snowden, we find that most of the time the NSA breaks cryptography by circumventing it. The NSA hacks the computers doing the encryption and decryption. It exploits bad implementations. It exploits weak or default keys. Or it "exfiltrates"—NSA-speak for steals—keys. Yes, it has some mathematics that we don't know about, but that's the exception. The most amazing thing about the NSA as revealed by Snowden is that it isn't made of magic.

This doesn't mean that cryptography is useless: far from it. What cryptography does is raise both the cost and risk of attack. Data zipping around the Internet unencrypted can be collected wholesale with minimal effort. Encrypted data has to be targeted individually. The NSA—or whoever is after your data—needs to target you individually and attack your computer and network specifically. That takes time and manpower, and is inherently risky. No organization has enough budget to do that to everyone; they have to pick and choose. While ubiquitous encryption won't eliminate targeted collection, it does have the potential to make bulk collection infeasible. The goal is to leverage the economics, the physics, and the math.

There's one more problem, though—one that the Snowden documents have illustrated well. Yes, technology can trump politics, but politics can also trump technology. Governments can use laws to subvert cryptography. They can sabotage the cryptographic standards in the communications and computer systems you use. They can deliberately insert backdoors into those same systems. They can do all of those, and then forbid the corporations implementing those systems to tell you about it. We know the NSA does this; we have to assume that other governments do the same thing.

Never forget, though, that while cryptography is still an essential tool for security, cryptography does not automatically mean security. The technical challenges of implementing cryptography are far more difficult than the mathematical challenges

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

of making the cryptography secure. And remember that the political challenges of being able to implement strong cryptography are just as important as the technical challenges. Security is only as strong as the weakest link, and the further away you get from the mathematics, the weaker the links become.

The 1995 world of *Applied Cryptography*, Second Edition, was very different from today's world. That was a singular time in academic cryptography, when I was able to survey the entire field of research and put everything under one cover. Today, there's too much, and the task of compiling it all is just too great. For those who want a more current book, I recommend *Cryptography Engineering*, which I wrote in 2010 with Niels Ferguson and Tadayoshi Kohno. But for a review of those heady times of the mid-1990s, and an introduction to what has become an essential technology of the Internet, *Applied Cryptography* still holds up surprisingly well.

—Minneapolis, Minnesota, and Cambridge, Massachusetts, January 2015

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Foreword
By Whitfield Diffie

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

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

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

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

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

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

Whitfield Diffie
Mountain View, CA

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Preface

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

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

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

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

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

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

How to Read This Book

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

Chapter 1 introduces cryptography, defines many terms, and briefly discusses precomputer cryptography.

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

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

Security First Innovations, LLC, Exhibit 2010
Page 2010 - 20
IPR2024-00212, Google LLC v. Security First Innovations, LLC

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

Acknowledgments

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

Bruce Schneier

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

About the Author

BRUCE SCHNEIER is an internationally renowned security technologist, called a "security guru" by *The Economist*. He is the author of twelve books — including his seminal work, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, and *Secrets & Lies: Digital Security in a Networked World* which has become a classic as well as hundreds of articles, essays, and academic papers. His influential newsletter "Crypto-Gram" and blog "Schneier on Security" are read by over 250,000 people. Schneier is a fellow at the Berkman Center for Internet and Society at Harvard Law School, a program fellow at the New America Foundation's Open Technology Institute, a board member of the Electronic Frontier Foundation, and an Advisory Board member of the Electronic Privacy Information Center. He is also the Chief Technology Officer of Resilient Systems, Inc. You can read his blog, essays, and academic papers at `www.schneier.com`. He tweets at `@schneierblog`.

10.1002/9781119183471.fmatter, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.fmatter, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

CHAPTER 3

Basic Protocols

3.1 KEY EXCHANGE

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

Key Exchange with Symmetric Cryptography

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

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

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

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

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

Key Exchange with Public-Key Cryptography

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

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

Man-in-the-Middle Attack

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

Interlock Protocol

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

Key Exchange with Digital Signatures

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Key and Message Transmission

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

(1) Alice generates a random session key, K, and encrypts M using K.

$E_K(M)$

(2) Alice gets Bob's public key from the database.

(3) Alice encrypts K with Bob's public key.

$E_B(K)$

(4) Alice sends both the encrypted message and encrypted session key to Bob.

$E_K(M), E_B(K)$

For added security against man-in-the-middle attacks, Alice can sign the transmission.

(5) Bob decrypts Alice's session key, K, using his private key.

(6) Bob decrypts Alice's message using the session key.

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

Key and Message Broadcast

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

(1) Alice generates a random session key, K, and encrypts M using K.

$E_K(M)$

(2) Alice gets Bob's, Carol's, and Dave's public keys from the database.

(3) Alice encrypts K with Bob's public key, encrypts K with Carol's public key, and then encrypts K with Dave's public key.

$E_B(K), E_C(K), E_D(K)$

(4) Alice broadcasts the encrypted message and all the encrypted keys to anybody who cares to receive it.

$E_B(K), E_C(K), E_D(K), E_K(M)$

(5) Only Bob, Carol, and Dave can decrypt the key, K, each using his or her private key.

(6) Only Bob, Carol, and Dave can decrypt Alice's message using K.

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

3.2 AUTHENTICATION

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

Authentication Using One-Way Functions

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

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

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

Dictionary Attacks and Salt

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

SKEY

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

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

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

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

Authentication Using Public-Key Cryptography

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

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

Mutual Authentication Using the Interlock Protocol

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

SKID

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

(4) Alice sends Bob:

$$H_K(R_B,A)$$

A is Alice's name.

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

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

Message Authentication

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

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

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

3.3 AUTHENTICATION AND KEY EXCHANGE

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

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

Wide-Mouth Frog

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 3.1
Symbols used in authentication and key exchange protocols

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

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

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

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

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

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

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

Yahalom

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

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

$$A, R_A$$

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

Needham-Schroeder

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

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

A,B,R_A

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

$E_A(R_A,B,K,E_B(K,A))$

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

$E_B(K,A)$

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

$E_K(R_B)$

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

$E_K(R_B - 1)$

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

(1) Mallory sends Bob the following message:

$$E_B(K,A)$$

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

$$E_K(R_B)$$

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

$$E_K(R_B - 1)$$

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

Now, Mallory has Bob convinced that he is Alice.

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

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

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

Otway-Rees

This protocol also uses symmetric cryptography [1224].

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

Kerberos

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

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

$$A, B$$

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

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

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

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

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

$$E_K(T + 1)$$

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

Neuman-Stubblebine

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

A, R_A

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

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

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

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

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

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

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

$E_K(R'_B)$

The new random numbers prevent replay attacks.

DASS

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

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

B

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

$S_T(B, K_B)$

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

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

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

A

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

$S_T(A, K_A)$

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

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

$E_K(T_B)$

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Denning-Sacco

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

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

$$A,B$$

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

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

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

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

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

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

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

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

$$B,C$$

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

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

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

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

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

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

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

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

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

Woo-Lam

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

A,B

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

$S_T(K_B)$

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

$E_{K_B}(A,R_A)$

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

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

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

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

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

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

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

$E_K(R_B)$

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

Other Protocols

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

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

Lessons Learned

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

— Many protocols failed because the designers tried to be too clever. They optimized their protocols by leaving out important pieces: names, random numbers, and so on. The remedy is to make everything explicit [43,44].

— Trying to optimize is an absolute tar pit and depends a whole lot on the assumptions you make. For example: If you have authenticated time, you can do a whole lot of things you can't do if you don't.

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

3.4 Formal Analysis of Authentication and Key-Exchange Protocols

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

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

> Alice believes X. (Alice acts as though X is true.)
>
> Alice sees X. (Someone has sent a message containing X to Alice, who can read and repeat X—possibly after decrypting it.)
>
> Alice said X. (At some time, Alice sent a message that includes the statement X. It is not known how long ago the message was sent or even that it was sent during the current run of the protocol. It is known that Alice believed X when she said it.)
>
> X is fresh. (X has not been sent in a message at any time before the current run of the protocol.)

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

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

Another rule is the nonce-verification rule:

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

There are four steps in BAN analysis:

(1) Convert the protocol into idealized form, using the statements previously described.

(2) Add all assumptions about the initial state of the protocol.

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

(3) Attach logical formulas to the statements: assertions about the state of the system after each statement.

(4) Apply the logical postulates to the assertions and assumptions to discover the beliefs held by the parties in the protocol.

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

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

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

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

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

— Accept (Bob, Alice, M, N). (Bob accepts the message M as from Alice during Bob's local round N.)

— Learn (Eve, M). (Eve learns M.)

— Send (Alice, Bob, Q, M). (Alice sends M to Bob in response to query, Q.)

— Request (Bob, Alice, Q, N). (Bob sends Q to Alice during Bob's local round N.)

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

— If Bob accepted message M from Alice at some point in the past, then Eve did not learn M at some point in the past.

— If Bob accepted message M from Alice in Bob's local round N, then Alice sent M to Bob as a response to a query in Bob's local round N.

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

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

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

3.5 MULTIPLE-KEY PUBLIC-KEY CRYPTOGRAPHY

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

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

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

TABLE 3.2
Three-Key Key Distribution

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

Broadcasting a Message

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

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

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

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

TABLE 3.3
Three-Key Message Encryption

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3.6 SECRET SPLITTING

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

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

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

(1) Trent generates a random-bit string, R, the same length as the message, M.

(2) Trent XORs M with R to generate S.

$$M \oplus R = S$$

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

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

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

$$R \oplus S = M$$

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

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

(1) Trent generates three random-bit strings, R, S, and T, the same length as the message, M.

(2) Trent XORs M with the three strings to generate U:

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

3.7 Secret Sharing

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

Secret Sharing with Cheaters

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

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

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

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

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

Secret Sharing without Trent

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

10.1002/9781119182471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119182471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Sharing a Secret without Revealing the Shares

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

Verifiable Secret Sharing

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

Secret-Sharing Schemes with Prevention

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

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

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

Secret Sharing with Disenrollment

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

3.8 Cryptographic Protection of Databases

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

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

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

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

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

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

10.1002/9781119183471.ch3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch3 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

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

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 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-

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 triple-DES, but it could just as easily be any algorithm.

Let $E_K(X)$ be triple-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

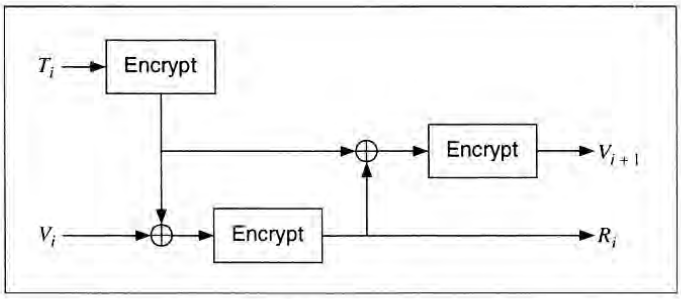

Figure 8.1 ANSI X9.17 key generation.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

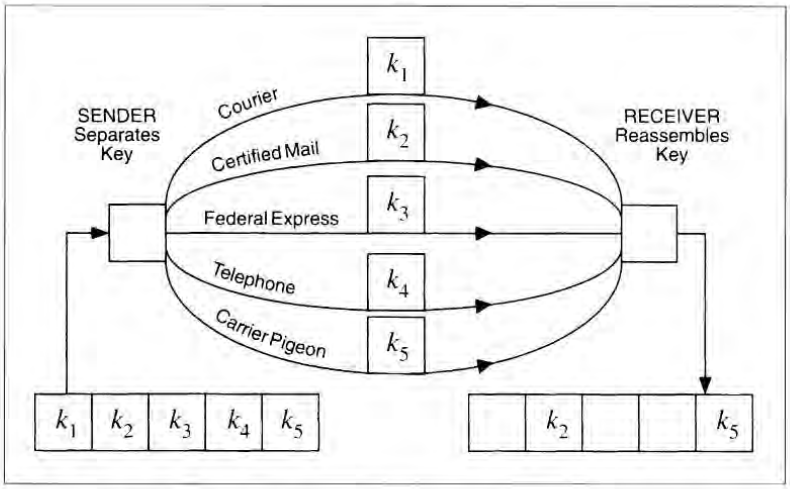

Figure 8.2 Key distribution via parallel channels.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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}.

Security First Innovations, LLC, Exhibit 2010
Page 2010 - 60
IPR2024-00212, Google LLC v. Security First Innovations, LLC

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 pseudorandom-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.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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?

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

— 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].

Security First Innovations, LLC, Exhibit 2010
Page 2010 - 66
IPR2024-00212, Google LLC v. Security First Innovations, LLC

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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.

10.1002/9781119183471.ch8, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch8 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

CHAPTER 15

Combining Block Ciphers

There are many ways to combine block algorithms to get new algorithms. The impetus behind these schemes is to try to increase security without going through the trouble of designing a new algorithm. DES is a secure algorithm; it has been cryptanalyzed for a good 20 years and the most practical way to break it is still brute force. However, the key is too short. Wouldn't it be nice to use DES as a building block for another algorithm with a longer key? We'd have the best of both worlds: the assurance of two decades of cryptanalysis plus a long key.

Multiple encryption is one combination technique: using an algorithm to encrypt the same plaintext block multiple times with multiple keys. Cascading is like multiple encryption, but uses different algorithms. There are other techniques.

Encrypting a plaintext block twice with the same key, whether with the same algorithm or a different one, is not smart. For the same algorithm, it does not affect the complexity of a brute-force search. (Remember, you assume a cryptanalyst knows the algorithm including the number of encryptions used.) For different algorithms, it may or may not. If you are going to use any of the techniques in this chapter, make sure the multiple keys are different and independent.

15.1 DOUBLE ENCRYPTION

A naïve way of improving the security of a block algorithm is to encrypt a block twice with two different keys. First encrypt a block with the first key, then encrypt the resulting ciphertext with the second key. Decryption is the reverse process.

$$C = E_{K_2}(E_{K_1}(P))$$
$$P = D_{K_1}(D_{K_2}(C))$$

If the block algorithm is a group (see Section 11.3), then there is always a K_3 such that

$$C = E_{K_2}(E_{K_1}(P)) = E_{K_3}(P)$$

If this is not the case, the resultant doubly-encrypted ciphertext block should be much harder to break using an exhaustive search. Instead of 2^n attempts (where n is the bit length of the key), it would require 2^{2n} attempts. If the algorithm is a 64-bit algorithm, the doubly-encrypted ciphertext would require 2^{128} attempts to find the key.

This turns out not to be true for a known-plaintext attack. Merkle and Hellman [1075] developed a time-memory trade-off that could break this double-encryption scheme in 2^{n+1} encryptions, not in 2^{2n} encryptions. (They showed this for DES, but the result can be generalized to any block algorithm.) The attack is called a **meet-in-the-middle attack**; it works by encrypting from one end, decrypting from the other, and matching the results in the middle.

In this attack, the cryptanalyst knows P_1, C_1, P_2, and C_2, such that

$$C_1 = E_{K_2}(E_{K_1}(P_1))$$
$$C_2 = E_{K_2}(E_{K_1}(P_2))$$

For each possible K, he computes $E_K(P_1)$ and stores the result in memory. After collecting them all, he computes $D_K(C_1)$ for each K and looks for the same result in memory. If he finds it, it is possible that the current key is K_2 and the key in memory is K_1. He tries encrypting P_2 with K_1 and K_2; if he gets C_2 he can be pretty sure (with a probability of success of 1 in 2^{2m-2n}, where m is the block size) that he has both K_1 and K_2. If not, he keeps looking. The maximum number of encryption trials he will probably have to run is $2 * 2^n$, or 2^{n+1}. If the probability of error is too large, he can use a third ciphertext block to get a probability of success of 1 in 2^{3m-2n}. There are still other optimizations [912].

This attack requires a lot of memory: 2^n blocks. For a 56-bit algorithm, this translates to 2^{56} 64-bit blocks, or 10^{17} bytes. This is still considerably more memory storage than one could comfortably comprehend, but it's enough to convince the most paranoid of cryptographers that double encryption is not worth anything.

For a 128-bit key, the amount of memory required is an enormous 10^{39} bytes. If we assume that a way exists to store a bit of information on a single atom of aluminum, the memory device required to launch this attack would be a cube of solid aluminum over a kilometer on a side. And then you need some place to put it! The meet-in-the middle attack seems infeasible for keys this size.

Another double-encryption method, sometimes called **Davies-Price**, is a variant of CBC [435].

$$C_i = E_{K_1}(P_i \oplus E_{K_2}(C_{i-1}))$$
$$P_i = D_{K_1}(C_i) \oplus E_{K_2}(C_{i-1})$$

They claim "no special virtue of this mode," but it seems to be vulnerable to the same meet-in-the-middle attacks as other double-encryption modes.

15.2 TRIPLE ENCRYPTION

Triple Encryption with Two Keys

A better idea, proposed by Tuchman in [1551], operates on a block three times with two keys: with the first key, then with the second key, and finally with the first key

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

again. He suggested that the sender first encrypt with the first key, then decrypt with the second key, and finally encrypt with the first key. The receiver decrypts with the first key, then encrypts with the second key, and finally decrypts with the first key.

$$C = E_{K_1}(D_{K_2}(E_{K_1}(P)))$$
$$P = D_{K_1}(E_{K_2}(D_{K_1}(C)))$$

This is sometimes called **encrypt-decrypt-encrypt (EDE)** mode [55]. If the block algorithm has an n-bit key, then this scheme has a $2n$-bit key. The curious encrypt-decrypt-encrypt pattern was designed by IBM to preserve compatibility with conventional implementations of the algorithm: Setting the two keys equal to each other is identical to encrypting once with the key. There is no security inherent in the encrypt-decrypt-encrypt pattern, but this mode has been adopted to improve the DES algorithm in the X9.17 and ISO 8732 standards [55,761].

K_1 and K_2 alternate to prevent the meet-in-the-middle attack previously described. If $C = E_{K_2}(E_{K_1}(E_{K_1}(P)))$, then a cryptanalyst could precompute $E_{K_1}(E_{K_1}(P)))$ for every possible K_1 and then proceed with the attack. It only requires 2^{n+2} encryptions.

Triple encryption with two keys is not susceptible to the same meet-in-the-middle attack described earlier. But Merkle and Hellman developed another time-memory trade-off that could break this technique in 2^{n-1} steps using 2^n blocks of memory [1075].

For each possible K_2, decrypt 0 and store the result in memory. Then, decrypt 0 with each possible K_1 to get P. Triple-encrypt P to get C, and then decrypt C with K_1. If that decryption is a decryption of 0 with a K_2 (stored in memory), the K_1 K_2 pair is a possible candidate. Check if it is right. If it's not, keep looking.

This is a chosen-plaintext attack, requiring an enormous amount of chosen plaintext to mount. It requires 2^n time and memory, and 2^m chosen plaintexts. It is not very practical, but it is a weakness.

Paul van Oorschot and Michael Wiener converted this to a known-plaintext attack, requiring p known plaintexts. This example assumes EDE mode.

(1) Guess the first intermediate value, a.

(2) Tabulate, for each possible K_1, the second intermediate value, b, when the first intermediate value is a, using known plaintext:

$$b = D_{K_1}(C)$$

where C is the resulting ciphertext from a known plaintext.

(3) Look up in the table, for each possible K_2, elements with a matching second intermediate value, b:

$$b = E_{K_2}(a)$$

(4) The probability of success is p/m, where p is the number of known plaintexts and m is the block size. If there is no match, try another a and start again.

The attack requires $2^{n+m}/p$ time and p memory. For DES, this is $2^{120}/p$ [1558]. For p greater than 256, this attack is faster than exhaustive search.

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Triple Encryption with Three Keys

If you are going to use triple encryption, I recommend three different keys. The key length is longer, but key storage is usually not a problem. Bits are cheap.

$$C = E_{K_3}(D_{K_2}(E_{K_1}(P)))$$
$$P = D_{K_1}(E_{K_2}(D_{K_3}(C)))$$

The best time-memory trade-off attack takes 2^{2n} steps and requires 2^n blocks of memory; it's a meet-in-the-middle attack [1075]. Triple encryption, with three independent keys, is as secure as one might naïvely expect double encryption to be.

Triple Encryption with Minimum Key (TEMK)

There is a secure way of using triple encryption with two keys that prevents the previous attack, called Triple Encryption with Minimum Key (TEMK) [858]. The trick is to derive three keys from two: X_1 and X_2:

$$K_1 = E_{X_1}(D_{X_2}(E_{X_1}(T_1)))$$
$$K_2 = E_{X_1}(D_{X_2}(E_{X_1}(T_2)))$$
$$K_3 = E_{X_1}(D_{X_2}(E_{X_1}(T_3)))$$

T_1, T_2, and T_3 are constants, which do not have to be secret. This is a special construction that guarantees that for any particular pair of keys, the best attack is a known-plaintext attack.

Triple-Encryption Modes

It's not enough to just specify triple encryption; there are several ways to do it. The decision of which to use affects both security and efficiency.

Here are two possible triple-encryption modes:

Inner-CBC: Encrypt the entire file in CBC mode three different times (see Figure 15.1a). This requires three different IVs.

$$C_i = E_{K_3}(S_i \oplus C_{i-1});\ S_i = D_{K_2}(T_i \oplus S_{i-1});\ T_i = E_{K_1}(P_i \oplus T_{i-1})$$
$$P_i = T_{i-1} \oplus D_{K_1}(T_i);\ T_i = S_{i-1} \oplus E_{K_2}(S_i);\ S_i = C_{i-1} \oplus D_{K_3}(C_i)$$

C_0, S_0, and T_0 are IVs.

Outer-CBC: Triple-encrypt the entire file in CBC mode (see Figure 15.1b). This requires one IV.

$$C_i = E_{K_3}(D_{K_2}(E_{K_1}(P_i \oplus C_{i-1})))$$
$$P_i = C_{i-1} \oplus D_{K_1}(E_{K_2}(D_{K_3}(C_i)))$$

Both modes require more resources than single encryption: more hardware or more time. However, given three encryption chips, the throughput of inner-CBC is no slower than single encryption. Since the three CBC encryptions are independent, three chips can be kept busy all the time, each feeding back into itself.

On the other hand, outer-CBC feedback is outside the three encryptions. This means that even with three chips, the throughput is only one-third that of single

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo, Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

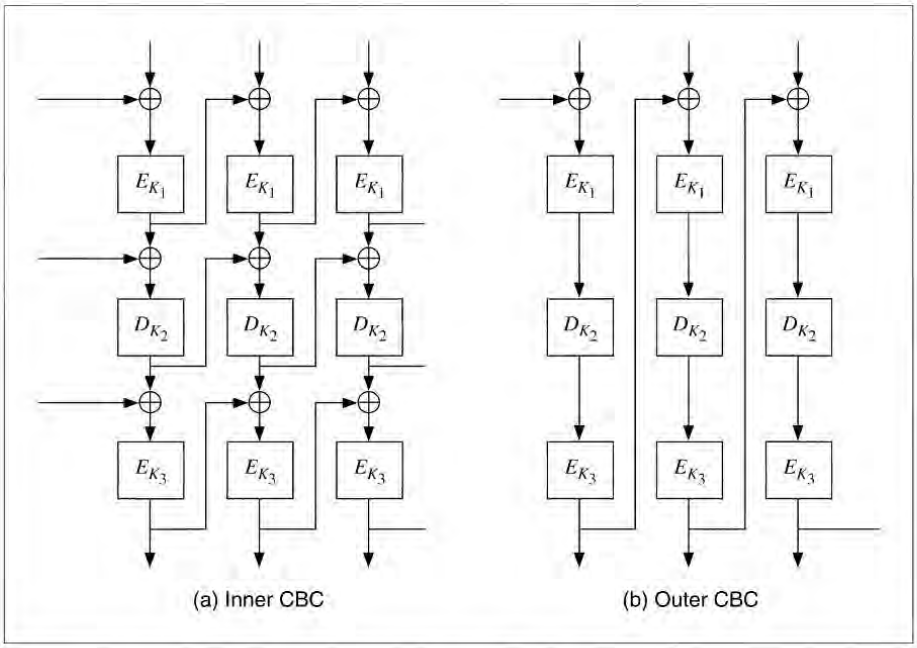

Figure 15.1 Triple encryption in CBC mode.

encryption. To get the same throughput with outer-CBC, you need to interleave IVs (see Section 9.12):

$$C_i = E_{K_3}(D_{K_2}(E_{K_1}(P_i \oplus C_{i-3})))$$

In this case C_0, C_{-1}, and C_{-2} are IVs. This doesn't help software implementations any, unless you have a parallel machine.

Unfortunately, the simpler mode is also the least secure. Biham analyzed various modes with respect to chosen-ciphertext differential cryptanalysis and found that inner-CBC is only slightly more secure than single encryption against a differential attack. If you think of triple encryption as a single larger algorithm, then inner feedbacks allow the introduction of external and known information into the inner workings of the algorithm; this facilitates cryptanalysis. The differential attacks require enormous amounts of chosen ciphertext to mount and are not very practical, but the results should be enough to give the most paranoid pause. Another analysis against meet-in-the-middle and brute-force attacks concludes that they are all equally secure [806].

There are other modes, as well. You can encrypt the entire file once in ECB, then twice in CBC; or once in CBC, once in ECB, and once in CBC; or twice in CBC and once in ECB. Biham showed that these variants are no more secure than single DES against a chosen-plaintext differential cryptanalysis attack [162]. And he doesn't

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

have high hopes for any other variants. If you are going to use triple encryption, use modes with outer feedback.

Variants on Triple Encryption

Before there were proofs that DES does not form a group, several schemes were proposed for multiple encryption. One way to guarantee that triple encryption doesn't reduce to single encryption is to change the effective block size. One simple method is to add a bit of padding. Pad the text with a string of random bits, half a block in length, between the first and second and between the second and third encryptions (see Figure 15.2). If p is the padding function, then:

$$C = E_{K_3}(p(E_{K_2}(p(E_{K_1}(P)))))$$

This padding not only disrupts patterns, but also overlaps encrypted blocks like bricks. It only adds one block to the length of the message.

Another technique, proposed by Carl Ellison, is to use some kind of keyless permutation function between the three encryptions. The permutation could work on large blocks—8 kilobytes or so—and would effectively give this variant a block size of 8 kilobytes. Assuming that the permutation is fast, this variant is not much slower than basic triple encryption.

$$C = E_{K_3}(T(E_{K_2}(T(E_{K_1}(P)))))$$

T collects a block of input (up to 8 kilobytes in length) and uses a pseudo-random-number generator to transpose it. A 1-bit change in the input causes 8 changed output bytes after the first encryption, up to 64 changed output bytes after the second encryption, and up to 512 changed output bytes after the third encryption. If each block algorithm is in CBC mode, as originally proposed, then the effect of a single

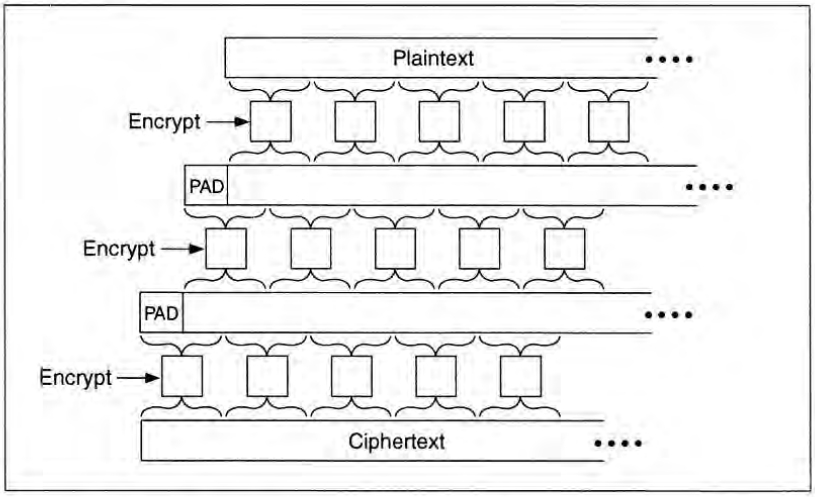

Figure 15.2 Triple encryption with padding.

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

changed input bit is likely to be the entire 8 kilobyte block, even in blocks other than the first.

The most recent variant of this scheme responded to Biham's attack on inner-CBC by including a whitening pass to hide plaintext patterns. That pass is a stream XOR with a cryptographically secure random-number generator called R below. The T on either side of it prevents the cryptanalyst from knowing *a priori* which key was used to encrypt any given byte on input to the last encryption. The second encryption is labelled nE (encryption with one of n different keys, used cyclically):

$$C = E_{K_3}(R(T(nE_{K_2}(T(E_{K_1}(R))))))$$

All encryptions are in ECB mode and keys are provided at least for the $n + 2$ encryption keys and the cryptographically secure random-number generator.

This scheme was proposed with DES, but works with any block algorithm. I know of no analysis of the security of this scheme.

15.3 Doubling the Block Length

There is some argument in the academic community whether a 64-bit block is long enough. On the one hand, a 64-bit block length only diffuses plaintext over 8 bytes of ciphertext. On the other hand, a longer block length makes it harder to hide patterns securely; there is more room to make mistakes.

Some propose doubling the block length of an algorithm using multiple encryptions [299]. Before implementing any of these, look for the possibility of meet-in-the-middle attacks. Richard Outerbridge's scheme [300], illustrated in Figure 15.3, is no more secure than single-block, two-key triple encryption [859].

However, I advise against this sort of thing. It isn't faster than conventional triple encryption: six encryptions are still required to encrypt two blocks of data. We know the characteristics of triple encryption; constructions like this often have hidden problems.

15.4 Other Multiple Encryption Schemes

The problem with two-key triple encryption is that it only doubles the size of the keyspace, but it requires three encryptions per block of plaintext. Wouldn't it be nice to find some clever way of combining two encryptions that would double the size of the keyspace?

Double OFB/Counter

This method uses a block algorithm to generate two keystreams, which are then used to encrypt the plaintext.

$$S_i = E_{K_1}(S_{i-1} \oplus I_1);\ I_1 = I_1 + 1$$
$$T_i = E_{K_2}(T_{i-1} \oplus I_2);\ I_2 = I_2 + 1$$
$$C_i = P_i \oplus S_i \oplus T_i$$

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

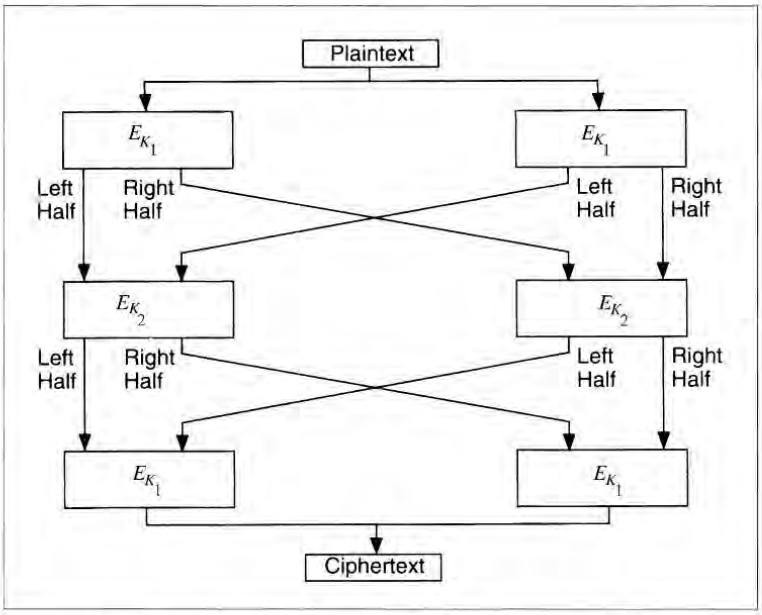

Figure 15.3 Doubling the block length.

S_i and T_i are internal variables, and I_1 and I_2 are counters. Two copies of the block algorithm run in a kind of hybrid OFB/counter mode, and the plaintext, S_i, and T_i are XORed together. The two keys, K_1 and K_2, are independent. I know of no cryptanalysis of this variant.

ECB + OFB

This method was designed for encrypting multiple messages of a fixed length, for example, disk blocks [186,188]. Use two keys: K_1 and K_2. First, use the algorithm and K_1 to generate a mask of the required block length. This mask will be used repeatedly to encrypt messages with the same keys. Then, XOR the plaintext message with the mask. Finally, encrypt the XORed plaintext with the algorithm and K_2 in ECB mode.

This mode has not been analyzed outside the paper in which it was proposed. Clearly it is at least as strong as a single ECB encryption and may be as strong as two passes with the algorithm. Possibly, a cryptanalyst could search for the two keys independently, if several known plaintext files are encrypted with the same key.

To thwart analysis of identical blocks in the same positions of different messages, you can add an IV. Unlike an IV in any other mode, here the IV is XORed with every block of the message before ECB encryption.

Matt Blaze designed this mode for his UNIX Cryptographic File System (CFS). It is a nice mode because the latency is only one encryption in ECB mode; the mask can be generated once and stored. In CFS, DES is the block algorithm.

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

xDES^i

In [1644,1645], DES is used as a building block for a series of block algorithms with both larger key sizes and larger block sizes. These constructions do not depend on DES in any way and can be used with any block algorithm.

The first, xDES^1, is simply a Luby-Rackoff construction with the block cipher as the underlying function (see Section 14.11). The block size is twice the size of the underlying block cipher and the key size is three times the size of the underlying block cipher. In each of 3 rounds, encrypt the right half with the block algorithm and one of the keys, XOR the result with the left half, and swap the two halves.

This is faster than conventional triple encryption, since three encryptions encrypt a block twice as large as the underlying algorithm. But there is also a simple meet-in-the-middle attack that finds the key with a table the size of 2^k, where k is the key size of the underlying algorithm. Encrypt the right half of a plaintext block with all possible values of K_1, XOR the left half of the plaintext, and store these values in a table. Then, encrypt the right half of the ciphertext with all possible values of K_3 and look for a match in the table. If you find one, the key pair K_1 and K_3 are possible candidates for the right key. Repeat the attack a few times, and only one candidate will remain. This shows that xDES^1 is not an ideal solution. Even worse, there is a chosen plaintext attack that proves xDES^1 is not much stronger than the underlying block cipher [858].

xDES^2 extends this idea to a 5-round algorithm with a block size 4 times that of the underlying block cipher and a key size 10 times that of the underlying block cipher. Figure 15.4 is one round of xDES^2; each of the four sub-blocks are the size of the underlying block ciphers and all 10 keys are independent.

This scheme is also faster than triple encryption: Ten encryptions are used to encrypt a block four times the size of the underlying block cipher. However, it is vulnerable to differential cryptanalysis [858] and should not be used. The scheme is even vulnerable if DES with independent round keys is used.

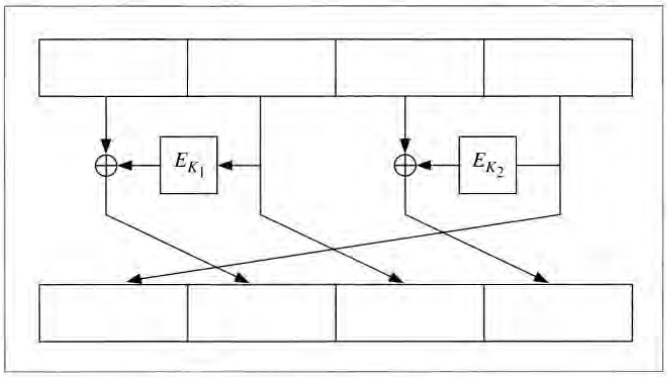

Figure 15.4 One round of xDES^2.

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

For $i \geq 3$, $x\text{DES}^i$ is probably too big to be useful as a block algorithm. For example, the block size for $x\text{DES}^3$ is 6 times that of the underlying cipher, the key size is 21 times, and 21 encryptions are required to encrypt a block 6 times that of the underlying block cipher. Triple encryption is faster.

Quintuple Encryption

If triple encryption isn't secure enough—perhaps you need to encrypt triple-encryption keys using an even stronger algorithm—then higher multiples might be in order. Quintuple encryption is very strong against meet-in-the-middle attacks. (Similar arguments to the ones used with double encryption can show that quadruple encryption provides minimal security improvements over triple encryption.)

$$C = E_{K_1}(D_{K_2}(E_{K_3}(D_{K_2}(E_{K_1}(P)))))$$
$$P = D_{K_1}(E_{K_2}(D_{K_3}(E_{K_2}(D_{K_1}(C)))))$$

This construction is backwards compatible with triple encryption if $K_2 = K_3$, and is backwards compatible with single encryption if $K_1 = K_2 = K_3$. Of course, it would be even stronger if all five keys were independent.

15.5 CDMF KEY SHORTENING

This method was designed by IBM for their Commercial Data Masking Facility or CDMF (see Section 24.8) to shrink a 56-bit DES key to a 40-bit key suitable for export [785]. It assumes that the original DES key includes the parity bits.

(1) Zero the parity bits: bits 8, 16, 24, 32, 40, 48, 56, 64.

(2) Encrypt the output of step (1) with DES and the key 0xc408b0540ba1e0ae, and XOR the result with the output of step (1).

(3) Take the output of step (2) and zero the following bits: 1, 2, 3, 4, 8, 16, 17, 18, 19, 20, 24, 32, 33, 34, 35, 36, 40, 48, 49, 50, 51, 52, 56, 64.

(4) Encrypt the output of step (3) with DES and the following key: 0xef2c041ce6382fe6. This key is then used for message encryption.

Remember that this method shortens the key length, and thereby weakens the algorithm.

15.6 WHITENING

Whitening is the name given to the technique of XORing some key material with the input to a block algorithm, and XORing some other key material with the output. This was first done in the DESX variant developed by RSA Data Security, Inc., and then (presumably independently) in Khufu and Khafre. (Rivest named this technique; it's a nonstandard usage of the word.)

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The idea is to prevent a cryptanalyst from obtaining a plaintext/ciphertext pair for the underlying algorithm. The technique forces a cryptanalyst to guess not only the algorithm key, but also one of the whitening values. Since there is an XOR both before and after the block algorithm, this technique is not susceptible to a meet-in-the-middle attack.

$$C = K_3 \oplus E_{K_2}(P \oplus K_1)$$
$$P = K_1 \oplus D_{K_2}(C \oplus K_3)$$

If $K_1 = K_3$, then a brute-force attack requires $2^{n+m/p}$ operations, where n is the key size, m is the block size, and p is the number of known plaintexts. If K_1 and K_3 are different, then a brute-force attack requires 2^{n+m+1} operations with three known plaintexts. Against differential and linear cryptanalysis, these measures only provide a few key bits of protection. But computationally this is a very cheap way to increase the security of a block algorithm.

15.7 Cascading Multiple Block Algorithms

What about encrypting a message once with Algorithm A and key K_A, then again with Algorithm B and key K_B? Maybe Alice and Bob have different ideas about which algorithms are secure: Alice wants to use Algorithm A and Bob wants to use Algorithm B. This technique is sometimes called **cascading**, and can be extended far beyond only two algorithms and keys.

Pessimists have said that there is no guarantee that the two algorithms will work together to increase security. There may be subtle interactions between the two algorithms that actually *decrease* security. Even triple encryption with three different algorithms may not be as secure as you think. Cryptography is a black art; if you don't know what you are doing, you can easily get into trouble.

Reality is much rosier. The previous warnings are true only if the different keys are related to each other. If all of the multiple keys are independent, then the resultant cascade is at least as difficult to break as the first algorithm in the cascade [1033]. If the second algorithm is vulnerable to a chosen-plaintext attack, then the first algorithm might facilitate that attack and make the second algorithm vulnerable to a known-plaintext attack when used in a cascade. This potential attack is not limited to encryption algorithms: If you let someone else specify any algorithm which is used on your message before encryption, then you had better be sure that your encryption will withstand a chosen-plaintext attack. (Note that the most common algorithm used for compressing and digitizing speech to modem speeds, used before any encryption, is CELP—designed by the NSA.)

This can be better phrased: Using a chosen-plaintext attack, a cascade of ciphers is at least as hard to break as any of its component ciphers [858]. A previous result showed that the cascade is at least as difficult to break as the strongest algorithm, but that result is based on some unstated assumptions [528]. Only if the algorithms commute, as they do in the case of cascaded stream ciphers (or block ciphers in OFB mode), is the cascade at least as strong as the strongest algorithm.

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

If Alice and Bob do not trust each other's algorithms, they can use a cascade. If these are stream algorithms, the order doesn't matter. If they are block algorithms, Alice can first use Algorithm A and then use Algorithm B. Bob, who trusts Algorithm B more, can use Algorithm B followed by Algorithm A. They might even add a good stream cipher between the two algorithms; it can't hurt and could very well increase security.

Remember that the keys for each algorithm in the cascade must be independent. If Algorithm A has a 64-bit key and Algorithm B has a 128-bit key, then the resultant cascade must have a 192-bit key. If you don't use independent keys, then the pessimists are much more likely to be right.

15.8 COMBINING MULTIPLE BLOCK ALGORITHMS

Here's another way to combine multiple block algorithms, one that is guaranteed to be at least as secure as both algorithms. With two algorithms (and two independent keys):

(1) Generate a random-bit string, R, the same size as the message M.
(2) Encrypt R with the first algorithm.
(3) Encrypt $M \oplus R$ with the second algorithm.
(4) The ciphertext message is the results of steps (2) and (3).

Assuming the random-bit string is indeed random, this method encrypts M with a one-time pad and then encrypts both the pad and the encrypted message with each of the two algorithms. Since both are required to reconstruct M, a cryptanalyst must break both algorithms. The drawback is that the ciphertext is twice the size of the plaintext.

This method can be extended to multiple algorithms, but the ciphertext expands with each additional algorithm. It's a good idea, but I don't think it's very practical.

10.1002/9781119183471.ch15, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/9781119183471.ch15 by Colette Woo , Wiley Online Library on [20/02/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License