Part II
Cryptographic Techniques

SECOND EDITION

APPLIED CRYPTOGRAPHY

Protocols, Algorithms, and Source Code in C

BRUCE SCHNEIER

disastrous impact on personal privacy to the high cost of having to add hardware to products that had previously encrypted in software. So far key escrow products are enjoying less than stellar sales and the scheme has attracted widespread negative comment, especially from the independent cryptographers. Some people, however, see more future in programming than politicking and have redoubled their efforts to provide the world with strong cryptography that is accessible to public scrutiny.

A sharp step back from the notion that export control law could supersede the First Amendment seemed to have been taken in 1980 when the *Federal Register* announcement of a revision to ITAR included the statement: "...provision has been added to make it clear that the regulation of the export of technical data does not purport to interfere with the First Amendment rights of individuals." But the fact that tension between the First Amendment and the export control laws has not gone away should be evident from statements at a conference held by RSA Data Security. NSA's representative from the export control office expressed the opinion that people who published cryptographic programs were "in a grey area" with respect to the law. If that is so, it is a grey area on which the first edition of this book has shed some light. Export applications for the book itself have been granted, with acknowledgement that published material lay beyond the authority of the Munitions Control Board. Applications to export the enclosed programs on disk, however, have been denied.

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

Whitfield Diffie
Mountain View, CA

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

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

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

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

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

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

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

As this is written, a new challenge confronts the public practice of cryptography. The government has augmented the widely published and available Data Encryption Standard, with a secret algorithm implemented in tamper–resistant chips. These chips will incorporate a codified mechanism of government monitoring. The negative aspects of this "key–escrow" program range from a potentially

Foreword By Whitfield Diffie

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

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

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

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

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

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

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

In my acceptance speech for the Donald E. Fink award—given for the best expository paper to

cryptographic algorithms in this book, get the disk. See the last page of the book for details.

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

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

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.

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

Preface

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

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

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

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

Do average people really need this kind of security? Yes. They may be planning a political campaign, discussing taxes, or having an illicit affair. They may be designing a new product, discussing a marketing strategy, or planning a hostile business takeover. Or they may be living in a country that does not respect the rights of privacy of its citizens. They may be doing something that they feel shouldn't be illegal, but is. For whatever reason, the data and communications are personal, private, and no one else's business.

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

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

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

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

About the Author

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

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 Cr• peau, 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, Ren• 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
Oak Park, Ill.
schneier@counterpane.com

24.14 Public-Key Cryptography Standards (PKCS)
24.15 Universal Electronic Payment System (UEPS)
24.16 Clipper
24.17 Capstone
24.18 AT&T Model 3600 Telephone Security Device (TSD)
Chapter 25—Politics
25.1 National Security Agency (NSA)
25.2 National Computer Security Center (NCSC)
25.3 National Institute of Standards and Technology (NIST)
25.4 RSA Data Security, Inc.
25.5 Public Key Partners
25.6 International Association for Cryptologic Research (IACR)
25.7 RACE Integrity Primitives Evaluation (RIPE)
25.8 Conditional Access for Europe (CAFE)
25.9 ISO/IEC 9979
25.10 Professional, Civil Liberties, and Industry Groups
25.11 Sci.crypt
25.12 Cypherpunks
25.13 Patents
25.14 U.S. Export Rules
25.15 Foreign Import and Export of Cryptography
25.16 Legal Issues
Afterword by Matt Blaze
Part V—Source Code
References
Index

20.1 Digital Signature Algorithm (DSA)
20.2 DSA Variants
20.3 Gost Digital Signature Algorithm
20.4 Discrete Logarithm Signature Schemes
20.5 Ong-Schnorr-Shamir
20.6 ESIGN
20.7 Cellular Automata
20.8 Other Public-Key Algorithms
Chapter 21—Identification Schemes
21.1 Feige-Fiat-Shamir
21.2 Guillou-Quisquater
21.3 Schnorr
21.4 Converting Identification Schemes to Signature Schemes
Chapter 22—Key-Exchange Algorithms
22.1 Diffie-Hellman
22.2 Station-to-Station Protocol
22.3 Shamir's Three-Pass Protocol
22.4 COMSET
22.5 Encrypted Key Exchange
22.6 Fortified Key Negotiation
22.7 Conference Key Distribution and Secret Broadcasting
Chapter 23—Special Algorithms for Protocols
23.1 Multiple-Key Public-Key Cryptography
23.2 Secret-Sharing Algorithms
23.3 Subliminal Channel
23.4 Undeniable Digital Signatures
23.5 Designated Confirmer Signatures
23.6 Computing with Encrypted Data
23.7 Fair Coin Flips
23.8 One-Way Accumulators
23.9 All-or-Nothing Disclosure of Secrets
23.10 Fair and Failsafe Cryptosystems
23.11 Zero-Knowledge Proofs of Knowledge
23.12 Blind Signatures
23.13 Oblivious Transfer
23.14 Secure Multiparty Computation
23.15 Probabilistic Encryption
23.16 Quantum Cryptography
Part IV—The Real World
Chapter 24—Example Implementations
24.1 IBM Secret-Key Management Protocol
24.2 MITRENET
24.3 ISDN
24.4 STU-III
24.5 Kerberos
24.6 KryptoKnight
24.7 SESAME
24.8 IBM Common Cryptographic Architecture
24.9 ISO Authentication Framework
24.10 Privacy-Enhanced Mail (PEM)
24.11 Message Security Protocol (MSP)
24.12 Pretty Good Privacy (PGP)
24.13 Smart Cards

16.3 Design and Analysis of Stream Ciphers
16.4 Stream Ciphers Using LFSRs
16.5 A5
16.6 Hughes XPD/KPD
16.7 Nanoteq
16.8 Rambutan
16.9 Additive Generators
16.10 Gifford
16.11 Algorithm M
16.12 PKZIP
Chapter 17—Other Stream Ciphers and Real Random-Sequence Generators
17.1 RC4
17.2 SEAL
17.3 WAKE
17.4 Feedback with Carry Shift Registers
17.5 Stream Ciphers Using FCSRs
17.6 Nonlinear-Feedback Shift Registers
17.7 Other Stream Ciphers
17.8 System-Theoretic Approach to Stream-Cipher Design
17.9 Complexity-Theoretic Approach to Stream-Cipher Design
17.10 Other Approaches to Stream-Cipher Design
17.11 Cascading Multiple Stream Ciphers
17.12 Choosing a Stream Cipher
17.13 Generating Multiple Streams from a Single Pseudo-Random-Sequence Generator
17.14 Real Random-Sequence Generators
Chapter 18—One-Way Hash Functions
18.1 Background
18.2 Snefru
18.3 N- Hash
18.4 MD4
18.5 MD5
18.6 MD2
18.7 Secure Hash Algorithm (SHA)
18.8 RIPE-MD
18.9 HAVAL
18.10 Other One-Way Hash Functions
18.11 One-Way Hash Functions Using Symmetric Block Algorithms
18.12 Using Public-Key Algorithms
18.13 Choosing a One-Way Hash Function
18.14 Message Authentication Codes
Chapter 19—Public-Key Algorithms
19.1 Background
19.2 Knapsack Algorithms
19.3 RSA
19.4 Pohlig-Hellman
19.5 Rabin
19.6 ElGamal
19.7 McEliece
19.8 Elliptic Curve Cryptosystems
19.9 LUC
19.10 Finite Automaton Public-Key Cryptosystems
Chapter 20—Public-Key Digital Signature Algorithms

Part III—Cryptographic Algorithms

Chapter 11—Mathematical Background

- 11.1 Information Theory
- 11.2 Complexity Theory
- 11.3 Number Theory
- 11.4 Factoring
- 11.5 Prime Number Generation
- 11.6 Discrete Logarithms in a Finite Field

Chapter 12—Data Encryption Standard (DES)

- 12.1 Background
- 12.2 Description of DES
- 12.3 Security of DES
- 12.4 Differential and Linear Cryptanalysis
- 12.5 The Real Design Criteria
- 12.6 DES Variants
- 12.7 How Secure Is DES Today?

Chapter 13—Other Block Ciphers

- 13.1 Lucifer
- 13.2 Madryga
- 13.3 NewDES
- 13.4 FEAL
- 13.5 REDOC
- 13.6 LOKI
- 13.7 Khufu and Khafre
- 13.8 RC2
- 13.9 IDEA
- 13.10 MMB
- 13.11 CA-1.1
- 13.12 Skipjack

Chapter 14—Still Other Block Ciphers

- 14.1 GOST
- 14.2 CAST
- 14.3 Blowfish
- 14.4 SAFER
- 14.5 3-Way
- 14.6 Crab
- 14.7 SXAL8/MBAL
- 14.8 RC5
- 14.9 Other Block Algorithms
- 14.10 Theory of Block Cipher Design
- 14.11 Using one-Way Hash Functions
- 14.12 Choosing a Block Algorithm

Chapter 15—Combining Block Ciphers

- 15.1 Double Encryption
- 15.2 Triple Encryption
- 15.3 Doubling the Block Length
- 15.4 Other Multiple Encryption Schemes
- 15.5 CDMF Key Shortening
- 15.6 Whitening
- 15.7 Cascading Multiple Block Algorithms
- 15.8 Combining Multiple Block Algorithms

Chapter 16—Pseudo-Random-Sequence Generators and Stream Ciphers

- 16.1 Linear Congruential Generators
- 16.2 Linear Feedback Shift Registers

5.7 Simultaneous Contract Signing
5.8 Digital Certified Mail
5.9 Simultaneous Exchange of Secrets
Chapter 6—Esoteric Protocols
6.1 Secure Elections
6.2 Secure Multiparty Computation
6.3 Anonymous Message Broadcast
6.4 Digital Cash

Part II—Cryptographic Techniques
Chapter 7—Key Length
7.1 Symmetric Key Length
7.2 Public-Key Key Length
7.3 Comparing Symmetric and Public-Key Key Length
7.4 Birthday Attacks against One-Way Hash Functions
7.5 How Long Should a Key Be?
7.6 Caveat Emptor
Chapter 8—Key Management
8.1 Generating Keys
8.2 Nonlinear Keyspaces
8.3 Transferring Keys
8.4 Verifying Keys
8.5 Using Keys
8.6 Updating Keys
8.7 Storing Keys
8.8 Backup Keys
8.9 Compromised Keys
8.10 Lifetime of Keys
8.11 Destroying Keys
8.12 Public-Key Key Management
Chapter 9—Algorithm Types and Modes
9.1 Electronic Codebook Mode
9.2 Block Replay
9.3 Cipher Block Chaining Mode
9.4 Stream Ciphers
9.5 Self-Synchronizing Stream Ciphers
9.6 Cipher-Feedback Mode
9.7 Synchronous Stream Ciphers
9.8 Output-Feedback Mode
9.9 Counter Mode
9.10 Other Block-Cipher Modes
9.11 Choosing a Cipher Mode
9.12 Interleaving
9.13 Block Ciphers versus Stream Ciphers
Chapter 10—Using Algorithms
10.1 Choosing an Algorithm
10.2 Public-Key Cryptography versus Symmetric Cryptography
10.3 Encrypting Communications Channels
10.4 Encrypting Data for Storage
10.5 Hardware Encryption versus Software Encryption
10.6 Compression, Encoding, and Encryption
10.7 Detecting Encryption
10.8 Hiding Ciphertext in Ciphertext
10.9 Destroying Information

Foreword by Whitfield Diffie
Preface
About the Author

Chapter 1—Foundations
- **1.1 Terminology**
- **1.2 Steganography**
- **1.3 Substitution Ciphers and Transposition Ciphers**
- **1.4 Simple XOR**
- **1.5 One-Time Pads**
- **1.6 Computer Algorithms**
- **1.7 Large Numbers**

Part I—Cryptographic Protocols

Chapter 2—Protocol Building Blocks
- **2.1 Introduction to Protocols**
- **2.2 Communications Using Symmetric Cryptography**
- **2.3 One-Way Functions**
- **2.4 One-Way Hash Functions**
- **2.5 Communications Using Public-Key Cryptography**
- **2.6 Digital Signatures**
- **2.7 Digital Signatures with Encryption**
- **2.8 Random and Pseudo-Random-Sequence Generation**

Chapter 3—Basic Protocols
- **3.1 Key Exchange**
- **3.2 Authentication**
- **3.3 Authentication and Key Exchange**
- **3.4 Formal Analysis of Authentication and Key-Exchange Protocols**
- **3.5 Multiple-Key Public-Key Cryptography**
- **3.6 Secret Splitting**
- **3.7 Secret Sharing**
- **3.8 Cryptographic Protection of Databases**

Chapter 4—Intermediate Protocols
- **4.1 Timestamping Services**
- **4.2 Subliminal Channel**
- **4.3 Undeniable Digital Signatures**
- **4.4 Designated Confirmer Signatures**
- **4.5 Proxy Signatures**
- **4.6 Group Signatures**
- **4.7 Fail-Stop Digital Signatures**
- **4.8 Computing with Encrypted Data**
- **4.9 Bit Commitment**
- **4.10 Fair Coin Flips**
- **4.11 Mental Poker**
- **4.12 One-Way Accumulators**
- **4.13 All-or-Nothing Disclosure of Secrets**
- **4.14 Key Escrow**

Chapter 5—Advanced Protocols
- **5.1 Zero-Knowledge Proofs**
- **5.2 Zero-Knowledge Proofs of Identity**
- **5.3 Blind Signatures**
- **5.4 Identity-Based Public-Key Cryptography**
- **5.5 Oblivious Transfer**
- **5.6 Oblivious Signatures**

Odds of drowning (in the U.S. per year)	1 in 59,000 (2^{16})
Odds of being killed in an automobile accident(in the U.S. in 1993)	1 in 6100 (2^{13})
Odds of being killed in an automobile accident(in the U.S. per lifetime)	1 in 88 (2^{7})
Time until the next ice age	14,000 (2^{14}) years
Time until the sun goes nova	10^{9} (2^{30}) years
Age of the planet	10^{9} (2^{30}) years
Age of the Universe	10^{10} (2^{34}) years
Number of atoms in the planet	10^{51} (2^{170})
Number of atoms in the sun	10^{57} (2^{190})
Number of atoms in the galaxy	10^{67} (2^{223})
Number of atoms in the Universe (dark matter excluded)	10^{77} (2^{265})
Volume of the Universe	10^{84} (2^{280}) cm^3
If the Universe is Closed:	
Total lifetime of the Universe	10^{11} (2^{37}) years 10^{18} (2^{61}) seconds
If the Universe is Open:	
Time until low-mass stars cool off	10^{14} (2^{47}) years
Time until planets detach from stars	10^{15} (2^{50}) years
Time until stars detach from galaxies	10^{19} (2^{64}) years
Time until orbits decay by gravitational radiation	10^{20} (2^{67}) years
Time until black holes decay by the Hawking process	10^{64} (2^{213}) years
Time until all matter is liquid at zero temperature	10^{65} (2^{216}) years
Time until all matter decays to iron	10^{1026} years
Time until all matter collapses to black holes	10^{1076} years

---

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

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

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

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

1.6 Computer Algorithms

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

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

These are the kinds of stuff this book is about.

1.7 Large Numbers

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

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

TABLE 1.1
Large Numbers

Physical Analogue	Number
Odds of being killed by lightning (per day)	1 in 9 billion (2^{33})
Odds of winning the top prize in a U.S. state lottery	1 in 4,000,000 (2^{22})
Odds of winning the top prize in a U.S. state lottery and being killed by lightning in the same day	1 in 2^{55}

O + T mod 26 = I
N + B mod 26 = P
E + F mod 26 = K
etc.

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

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

```
POYYAEAAZX
```

which would decrypt to:

```
SALMONEGGS
```

or

```
BXFGBMTMXM
```

which would decrypt to:

```
GREENFLUID
```

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

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

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

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

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

$P \oplus K = C$
$C \oplus K = P$

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

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

1. Discover the length of the key by a procedure known as **counting coincidences** [577]. XOR the ciphertext against itself shifted various numbers of bytes, and count those bytes that are equal. If the displacement is a multiple of the key length, then something over 6 percent of the bytes will be equal. If it is not, then less than 0.4 percent will be equal (assuming a random key encrypting normal ASCII text; other plaintext will have different numbers). This is called the **index of coincidence**. The smallest displacement that indicates a multiple of the key length is the length of the key.
2. Shift the ciphertext by that length and XOR it with itself. This removes the key and leaves you with plaintext XORed with the plaintext shifted the length of the key. Since English has 1.3 bits of real information per byte (see Section 11.1), there is plenty of redundancy for determining a unique decryption.

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

1.5 One-Time Pads

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

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

```
ONETIMEPAD
```

and the key sequence from the pad is

```
TBFRGFARFM
```

then the ciphertext is

```
IPKLPSFHGQ
```

because

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

1.4 Simple XOR

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

$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

Also note that:

$a \oplus a = 0$
$a \oplus b \oplus b = a$

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

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

void main (int argc, char *argv[])

{
     FILE *fi, *fo;
     char *cp;
     int c;

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

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

There are even more complicated transposition ciphers, but computers can break almost all of them.

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

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

Rotor Machines

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

A **rotor machine** has a keyboard and a series of rotors, and implements a version of the Vigenère cipher. Each rotor is an arbitrary permutation of the alphabet, has 26 positions, and performs a simple substitution. For example, a rotor might be wired to substitute "F" for "A," "U" for "B," "L" for "C," and so on. And the output pins of one rotor are connected to the input pins of the next.

Figure 1.4 *Columnar transposition cipher.*

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

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

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

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

Further Reading

P = ROT13 (ROT13 (P))

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

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

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

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

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

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

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

Transposition Ciphers

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

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

appreciably—most graphics standards specify more gradations of color than the human eye can notice—and the message can be stripped out at the receiving end. You can store a 64-kilobyte message in a 1024 × 1024 grey-scale picture this way. Several public-domain programs do this sort of thing.

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

1.3 Substitution Ciphers and Transposition Ciphers

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

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

Substitution Ciphers

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

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

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

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

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

Encrypting a file twice with ROT13 restores the original file.

plaintext is meaningful. This is called a **brute-force** attack (see Section 7.1).

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

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

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

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

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

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

Historical Terms

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

1.2 Steganography

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

More recently, people are hiding secret messages in graphic images. Replace the least significant bit of each byte of the image with the bits of the message. The graphical image won't change

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

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

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

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

Security of Algorithms

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

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

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

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

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

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

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

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

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

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

Given: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2),...P_i, C_i = E_k(P_i)$, where the cryptanalyst gets to choose $P_1, P_2,...P_i$

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

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

There are at least three other types of cryptanalytic attack.

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

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

Deduce: *k*

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

6. Chosen-key attack. This attack doesn't mean that the cryptanalyst can choose the key; it means that he has some knowledge about the relationship between different keys. It's strange and obscure, not very practical, and discussed in Section 12.4.

7. Rubber-hose cryptanalysis. The cryptanalyst threatens, blackmails, or tortures someone until they give him the key. Bribery is sometimes referred to as a **purchase-key attack**. These are all very powerful attacks and often the best way to break an algorithm.

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

called "public-key" because the encryption key can be made public: A complete stranger can use the encryption key to encrypt a message, but only a specific person with the corresponding decryption key can decrypt the message. In these systems, the encryption key is often called the **public key**, and the decryption key is often called the **private key**. The private key is sometimes also called the secret key, but to avoid confusion with symmetric algorithms, that tag won't be used here.

Encryption using public key K is denoted by:

$$E_K(M) = C$$

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

$$D_K(C) = M$$

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

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

Cryptanalysis

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

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

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

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

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

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

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

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

Figure 1.2 *Encryption and decryption with a key.*

Figure 1.3 *Encryption and decryption with two different keys.*

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

Symmetric Algorithms

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

Encryption and decryption with a symmetric algorithm are denoted by:

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

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

Public-Key Algorithms

Public-key algorithms (also called asymmetric algorithms) are designed so that the key used for encryption is different from the key used for decryption. Furthermore, the decryption key cannot (at least in any reasonable amount of time) be calculated from the encryption key. The algorithms are

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

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

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

Algorithms and Keys

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

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

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

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

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

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

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

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

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

Chapter 1
Foundations

1.1 Terminology

Sender and Receiver

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

Messages and Encryption

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

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

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

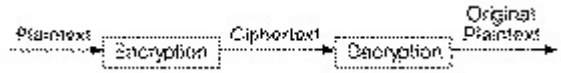

Figure 1.1 *Encryption and Decryption.*

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

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

$$E(M) = C$$

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

$$D(C) = M$$

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

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

Authentication, Integrity, and Nonrepudiation

Part I
Cryptographic protocols

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

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

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

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

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

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

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

Cryptographically Secure Pseudo-Random Sequences

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

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

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

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

Real Random Sequences

Now we're drifting into the domain of philosophers. Is there such a thing as randomness? What is a random sequence? How do you know if a sequence is random? Is "101110100" more random than "101010101"? Quantum mechanics tells us that there is honest-to-goodness randomness in the real world. But can we preserve that randomness in the deterministic world of computer chips and finite-state machines?

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

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

Even if the public keys are stored in a secure database, Mallory could still substitute one for another during transmission. To prevent this, Trent can sign each public key with his own private key. Trent, when used in this manner, is often known as a **Key Certification Authority** or **Key Distribution Center** (**KDC**). In practical implementations, the KDC signs a compound message consisting of the user's name, his public key, and any other important information about the user. This signed compound message is stored in the KDC's database. When Alice gets Bob's key, she verifies the KDC's signature to assure herself of the key's validity.

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

2.8 Random and Pseudo-Random-Sequence Generation

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

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

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

Pseudo-Random Sequences

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

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

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

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

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

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

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

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

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

Foiling the Resend Attack

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

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

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

Attacks against Public-Key Cryptography

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

2.7 Digital Signatures with Encryption

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

(1) Alice signs the message with her private key.
$S_A(M)$
(2) Alice encrypts the signed message with Bob's public key and sends it to Bob.
$E_B(S_A(M))$
(3) Bob decrypts the message with his private key.
$D_B(E_B(S_A(M))) = S_A(M)$
(4) Bob verifies with Alice's public key and recovers the message.
$V_A(S_A(M)) = M$

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

In electronic correspondence as well, signing before encrypting is a prudent practice [48]. Not only is it more secure—an adversary can't remove a signature from an encrypted message and add his own—but there are legal considerations: If the text to be signed is not visible to the signer when he affixes his signature, then the signature may have little legal force [1312]. And there are some cryptanalytic attacks against this technique with RSA signatures (see Section 19.3).

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

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

Resending the Message as a Receipt

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

(1) Alice signs a message with her private key, encrypts it with Bob's public key, and sends it to Bob.
$E_B(S_A(M))$
(2) Bob decrypts the message with his private key and verifies the signature with Alice's public key, thereby verifying that Alice signed the message and recovering the message.
$V_A(D_B(E_B(S_A(M)))) = M$
(3) Bob signs the message with his private key, encrypts it with Alice's public key, and sends it back to Alice.
$E_A(S_B(M))$
(4) Alice decrypts the message with her private key and verifies the signature with Bob's public key. If the resultant message is the same one she sent to Bob, she knows that Bob received the message accurately.

(4) Alice sends the document, her signature, and Bob's signature to Carol.
(5) Carol verifies both Alice's signature and Bob's signature.

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

Nonrepudiation and Digital Signatures

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

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

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

The general protocol is given in [28]:

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

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

Applications of Digital Signatures

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

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

bit hash are only one in 2^{160}, anyone can safely equate a signature of the hash with a signature of the document. If a non-one-way hash function were used, it would be an easy matter to create multiple documents that hashed to the same value, so that anyone signing a particular document would be duped into signing a multitude of documents.

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

Algorithms and Terminology

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

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

$$S_K(M)$$

and verifying a signature with the corresponding public key is:

$$V_K(M)$$

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

Multiple Signatures

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

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

(1) Alice signs the hash of the document.
(2) Bob signs the hash of the document.
(3) Bob sends his signature to Alice.

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

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

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

1. The signature is authentic; when Bob verifies the message with Alice's public key, he knows that she signed it.
2. The signature is unforgeable; only Alice knows her private key.
3. The signature is not reusable; the signature is a function of the document and cannot be transferred to any other document.
4. The signed document is unalterable; if there is any alteration to the document, the signature can no longer be verified with Alice's public key.
5. The signature cannot be repudiated. Bob doesn't need Alice's help to verify her signature.

Signing Documents and Timestamps

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

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

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

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

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

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

Speed increases drastically and, since the chances of two different documents having the same 160-

4. The signed document is unalterable. Were Bob to try to alter the document after receipt, Trent could prove foul play in exactly the same manner just described.
5. The signature cannot be repudiated. Even if Alice later claims that she never sent the message, Trent's certification says otherwise. Remember, Trent is trusted by everyone; what he says is true.

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

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

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

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

Digital Signature Trees

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

Signing Documents with Public-Key Cryptography

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

The basic protocol is simple:

5. The signature cannot be repudiated. The signature and the document are physical things. The signer cannot later claim that he or she didn't sign it.

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

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

Signing Documents with Symmetric Cryptosystems and an Arbitrator

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

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

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

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

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

1. This signature is authentic. Trent is a trusted arbitrator and Trent knows that the message came from Alice. Trent's certification serves as proof to Bob.
2. This signature is unforgeable. Only Alice (and Trent, but everyone trusts him) knows K_A, so only Alice could have sent Trent a message encrypted with K_A. If someone tried to impersonate Alice, Trent would have immediately realized this in step (2) and would not certify its authenticity.
3. This signature is not reusable. If Bob tried to take Trent's certification and attach it to another message, Alice would cry foul. An arbitrator (it could be Trent or it could be a completely different arbitrator with access to the same information) would ask Bob to produce both the message and Alice's encrypted message. The arbitrator would then encrypt the message with K_A and see that it did not match the encrypted message that Bob gave him. Bob, of course, could not produce an encrypted message that matches because he does not know K_A.

ever gets her hands on it, she can decrypt messages encrypted with it. With the previous protocol, the session key is created when it is needed to encrypt communications and destroyed when it is no longer needed. This drastically reduces the risk of compromising the session key. Of course, the private key is vulnerable to compromise, but it is at less risk because it is only used once per communication to encrypt a session key. This is further discussed in Section 3.1.

Merkle's Puzzles

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

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

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

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

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

2.6 Digital Signatures

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

1. The signature is authentic. The signature convinces the document's recipient that the signer deliberately signed the document.
2. The signature is unforgeable. The signature is proof that the signer, and no one else, deliberately signed the document.
3. The signature is not reusable. The signature is part of the document; an unscrupulous person cannot move the signature to a different document.
4. The signed document is unalterable. After the document is signed, it cannot be altered.

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

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

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

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

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

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

> **(1)** Bob sends Alice his public key.
> **(2)** Alice generates a random session key, K, encrypts it using Bob's public key, and sends it to Bob.
> $E_B(K)$
> **(3)** Bob decrypts Alice's message using his private key to recover the session key.
> $D_B(E_B(K)) = K$
> **(4)** Both of them encrypt their communications using the same session key.

Using public-key cryptography for key distribution solves a very important key-management problem. With symmetric cryptography, the data encryption key sits around until it is used. If Eve

block encryption algorithm; there are also dedicated MACs.

2.5 Communications Using Public-Key Cryptography

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

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

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

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

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

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

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

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

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

Hybrid Cryptosystems

easy to compute in one direction and hard to compute in the other direction. But, if you know the secret, you can easily compute the function in the other direction. That is, it is easy to compute $f(x)$ given x, and hard to compute x given $f(x)$. However, there is some secret information, y, such that given $f(x)$ and y it is easy to compute x.

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

2.4 One-Way Hash Functions

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

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

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

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

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

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

Message Authentication Codes

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

replacement message that would decrypt to gibberish. Bob, thinking the message came from Alice, might conclude that either the network or Alice had some serious problems.

What about Alice? What can she do to disrupt the protocol? She can give a copy of the key to Eve. Now Eve can read whatever Bob says. She can reprint his words in *The New York Times*. Although serious, this is not a problem with the protocol. There is nothing to stop Alice from giving Eve a copy of the plaintext at any point during the protocol. Of course, Bob could also do anything that Alice could. This protocol assumes that Alice and Bob trust each other.

In summary, symmetric cryptosystems have the following problems:

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

2.3 One-Way Functions

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

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

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

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

So, what good are one-way functions? We can’t use them for encryption as is. A message encrypted with the one-way function isn’t useful; no one could decrypt it. (Exercise: Write a message on a plate, smash the plate into tiny bits, and then give the bits to a friend. Ask your friend to read the message. Observe how impressed he is with the one-way function.) For public-key cryptography, we need something else (although there are cryptographic applications for one-way functions—see Section 3.2).

A **trapdoor one-way function** is a special type of one-way function, one with a secret trapdoor. It is

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

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

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

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

2.2 Communications Using Symmetric Cryptography

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

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

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

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

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

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

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

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

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

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

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

Self-Enforcing Protocols

A **self-enforcing protocol** is the best type of protocol. The protocol itself guarantees fairness (see Figure 2.1c). No arbitrator is required to complete the protocol. No adjudicator is required to resolve disputes. The protocol is constructed so that there cannot be any disputes. If one of the parties tries to cheat, the other party immediately detects the cheating and the protocol stops. Whatever the cheating party hoped would happen by cheating, doesn't happen.

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

Attacks against Protocols

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

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

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

Passive attackers try to gain information about the parties involved in the protocol. They collect

(4) Alice deposits the check.

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

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

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

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

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

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

Adjudicated Protocols

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

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

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

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

Nonarbitrated subprotocol (executed every time):

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

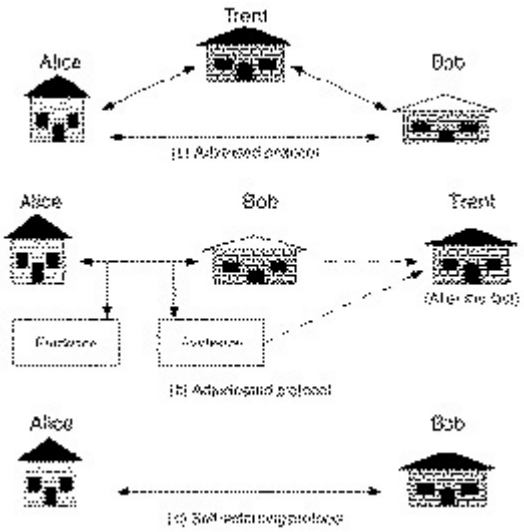

Figure 2.1 *Types of protocols.*

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

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

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

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

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

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

insecurity than it is to prove security.

The Purpose of Protocols

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

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

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

It is naïve to assume that people on computer networks are honest. It is naïve to assume that the managers of computer networks are honest. It is even naïve to assume that the designers of computer networks are honest. Most are, but the dishonest few can do a lot of damage. By formalizing protocols, we can examine ways in which dishonest parties can subvert them. Then we can develop protocols that are immune to that subversion.

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

The Players

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

Arbitrated Protocols

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

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

TABLE 2.1
Dramatis Personae

Chapter 2
Protocol Building Blocks

2.1 Introduction to Protocols

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

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

Protocols have other characteristics as well:

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

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

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

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

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

"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 communicate 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.

together, cannot prove that Carol's shadow is invalid.

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

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

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

Secret Sharing without Trent

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

Sharing a Secret without Revealing the Shares

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

Verifiable Secret Sharing

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

Secret-Sharing Schemes with Prevention

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

The math is complicated, but the basic idea is that everyone gets two shares: a "yes" share and a "no" share. When it comes time to reconstruct the secret, people submit one of their shares. The actual share they submit depends on whether they wish the secret reconstructed. If there are *m* or more

M. Remember, *M* isn't being split in the normal sense of the word; it is being XORed with random values.

3.7 Secret Sharing

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

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

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

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

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

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

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

Secret Sharing with Cheaters

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

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

tell only your most trusted employees the exact mixture of ingredients, but what if one of them defects to the competition? There goes the secret, and before long every grease palace on the block will be making burgers with sauce as tasteless as yours.

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

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

(1) Trent generates a random-bit string, R, the same length as the message, M.
(2) Trent XORs M with R to generate S.
$$M \oplus R = S$$
(3) Trent gives R to Alice and S to Bob.

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

(4) Alice and Bob XOR their pieces together to reconstruct the message:
$$R \oplus S = M$$

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

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

(1) Trent generates three random-bit strings, R, S, and T, the same length as the message, M.
(2) Trent XORs M with the three strings to generate U:
$$M \oplus R \oplus S \oplus T = U$$
(3) Trent gives R to Alice, S to Bob, T to Carol, and U to Dave.

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

(4) Alice, Bob, Carol, and Dave get together and compute:
$$R \oplus S \oplus T \oplus U = M$$

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

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

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 $_KB$	K_C
K_A and K_C	K_B
K_B and K_C	K_A

3.6 Secret Splitting

Imagine that you've invented a new, extra gooey, extra sweet, cream filling or a burger sauce that is even more tasteless than your competitors'. This is important; you have to keep it secret. You could

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 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.

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

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

Another rule is the nonce-verification rule:

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

There are four steps in BAN analysis:

> **(1)** Convert the protocol into idealized form, using the statements previously described.
> **(2)** Add all assumptions about the initial state of the protocol.
> **(3)** Attach logical formulas to the statements: assertions about the state of the system after each statement.
> **(4)** Apply the logical postulates to the assertions and assumptions to discover the beliefs held by the parties in the protocol.

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

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

Other logic systems have been published, some designed as extensions to BAN logic [645,586,1556,828] and others based on BAN to correct perceived weaknesses [1488,1002]. The most successful of these is GNY [645], although it has some shortcomings [40]. 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 toquery, *Q*.)
> — Request (Bob, Alice, *Q*, *N*). (Bob sends *Q* to Alice during Bob's local round *N*.)

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 developing 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:

Trent's public key, K_T.

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

(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.

$ST(KA),EKB(ST(RA,K,A,B))$

(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.

$E_{KA}(S_T(RA,K,A,B),R_B)$

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

$E_K(R_B)$

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

Other Protocols

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

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

Lessons Learned

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

— Many protocols failed because the designers tried to be too clever. They optimized their protocols by leaving out important pieces: names, random numbers, and so on. The remedy is to make everything explicit [43,44].
— Trying to optimize is an absolute tar pit and depends a whole lot on the assumptions you make. For example: If you have authenticated time, you can do a whole lot of things you can't do if you don't.
— The protocol of choice depends on the underlying communications architecture. Do you want to minimize the size of messages or the number of messages? Can all parties talk with each other or can only a few of them?

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

3.4 Formal Analysis of Authentication and Key-Exchange Protocols

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

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

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

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

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

(3) Alice sends Bob a random session key and a timestamp, signed in her private key and encrypted in Bob's public key, along with both signed public keys.
$E_B(S_A(K,T_A)),S_T(B,K_B),S_T(A,K_A)$

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

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

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

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

(2) Trent sends Bob both Bob's and Carol's signed public keys.
$S_T(B,K_B),S_T(C,K_C)$

(3) Bob sends Carol the signed session key and timestamp he previously received from Alice, encrypted with Carol's public key, along with Alice's certificate and Carol's certificate.
$E_C(S_A(K,T_A)),S_T(A,K_A),S_T(C,K_C)$

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

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

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

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

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

Woo-Lam

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

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

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

(3) Alice verifies Trent's signature. Then she sends Bob her name and a random number, encrypted with Bob's public key.
$E_{KB}(A,R_A)$

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

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

(1) Alice sends Bob the message Trent sent her in step (3) and a new random number.
$E_B(A,K,T_B),R'_A$

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

(3) Alice sends Bob his new random number, encrypted in their session key.
$E_K(R'_B)$

The new random numbers prevent replay attacks.

DASS

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

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

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

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

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

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

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

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

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

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

Denning-Sacco

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.

$EA(T,L,K,B),EB(T,L,K,A)$

(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 [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.

(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 time-stamp 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:
$I,A,B,E_A(R_A,\mathrm{I},\mathrm{A},\mathrm{B})$

(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:
$I,A,B,E_A(R_A,\mathrm{I},\mathrm{A},\mathrm{B}),E_B(R_B,\mathrm{I},\mathrm{A},\mathrm{B})$

(3) Trent generates a random session key. Then he creates two messages. One is Alice's random number and the session key, encrypted in the key he shares with Alice. The other is Bob's random number and the session key, encrypted in the key he shares with Bob. He sends these two messages, along with the index number, to Bob:
$I,E_A(R_A,K),E_B(R_B,\mathrm{K})$

(4) Bob sends Alice the message encrypted in her key, along with the index number:
$I,E_A(R_A,\mathrm{K})$

(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.

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 RA is the same value that she sent Trent in step (1). Then she sends Bob the message that Trent encrypted in his key.

$E_B(K,A)$

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

$E_K(R_B)$

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

$E_K(R_B - 1)$

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

All of this fussing around with R_A and R_B and R_B - 1 is to prevent **replay attacks**. In this attack, Mallory can record old messages and then use them later in an attempt to subvert the protocol. The presence of R_A in step (2) assures Alice that Trent's message is legitimate and not a replay of a response from a previous execution of the protocol. When Alice successfully decrypts R_B and sends Bob R_B - 1 in step (5), Bob is ensured that Alice's messages are not replays from an earlier execution of the protocol.

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

(1) Mallory sends Bob the following message:

$E_B(K,A)$

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

$E_K(R_B)$

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 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:

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

(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.

$$A,E_A(T_A,\mathrm{B},\mathrm{K})$$

(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:

$$E_B(T_B,\mathrm{A},\mathrm{K})$$

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.

$$B,E_B(A,R_A,\mathrm{R_B})$$

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

(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:

$$R_B, H_K(R_A, RB, _B)$$

H_K is the MAC. (The RIPE document suggests the RIPE-MAC function—see Section 18.14.)
B is Bob's name.
(3) Alice computes $H_K(R_A, R_B, B)$ and compares it with what she received from Bob. If the results are identical, then Alice knows that she is communicating with Bob.

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

(4) Alice sends Bob:

$$H_K(R_B, A)$$

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

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

Message Authentication

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

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

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

3.3 Authentication and Key Exchange

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

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

Wide-Mouth Frog

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

Mutual Authentication Using the Interlock Protocol

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

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

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

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

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

Donald Davies and 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:

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 na•ve attempt at a protocol. When logging in, the protocol proceeds as follows:

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

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

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

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

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

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

(3) The host compares the result of the one-way function to the value it previously stored.

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

Dictionary Attacks and Salt

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

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

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

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

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

SKEY

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

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

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

$E_K(M)$

(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 particular 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.

To see how this causes a problem for Mallory, let's review his attempt to subvert the protocol. He can still substitute his own public keys for Alice's and Bob's in steps (1) and (2). But now, when he intercepts half of Alice's message in step (3), he cannot decrypt it with his private key and re-encrypt it with Bob's public key. He must invent a totally new message and send half of it to Bob. When he intercepts half of Bob's message to Alice in step (4), he has the same problem. He cannot decrypt it with his private key and re-encrypt it with Alice's public key. He has to invent a totally new message and send half of it to Alice. By the time he intercepts the second halves of the real messages in steps (5) and (6), it is too late for him to change the new messages he invented. The conversation between Alice and Bob will necessarily be completely different.

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

Key Exchange with Digital Signatures

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

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

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

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

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

Key and Message Transmission

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

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

attack on the ciphertext, Mallory is a lot more powerful than Eve. Not only can he listen to messages between Alice and Bob, he can also modify messages, delete messages, and generate totally new ones. Mallory can imitate Bob when talking to Alice and imitate Alice when talking to Bob. Here's how the attack works:

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

Even if Alice's and Bob's public keys are stored on a database, this attack will work. Mallory can intercept Alice's database inquiry and substitute his own public key for Bob's. He can do the same to Bob and substitute his own public key for Alice's. Or better yet, he can break into the database surreptitiously and substitute his key for both Alice's and Bob's. Then he simply waits for Alice and Bob to talk with each other, intercepts and modifies the messages, and he has succeeded.

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

Interlock Protocol

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

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

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

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

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

What if a major key-escrow service, either government or commercial, had its entire escrowed key database stolen? What if the U.S. government tried to keep this quiet for a while? Clearly, this would have an impact on users' willingness to use key escrow. If it's not voluntary, a couple of scandals like this would increase political pressure to either make it voluntary, or to add complex new regulations to the industry.

Even more dangerous is a scandal where it becomes public that political opponent of the current administration, or some outspoken critic of some intelligence or police agencies has been under surveillance for years. This could raise public sentiment strongly against escrowed encryption.

If signature keys are escrowed as well as encryption keys, there are additional issues. Is it acceptable for the authorities to use signature keys to run operations against suspected criminals? Will the authenticity of signatures based on escrowed keys be accepted in courts? What recourse do users have if the authorities actually do use their signature keys to sign some unfavorable contract, to help out a state-supported industry, or just to steal money?

The globalization of cryptography raises an additional set of questions. Will key-escrow policies be compatible across national borders? Will multi-national corporations have to keep separate escrowed keys in every country to stay in compliance with the various local laws? Without some kind of compatibility, one of the supposed advantages of key-escrow schemes (international use of strong encryption) falls apart.

What if some countries don't accept the security of escrow agencies on faith? How do users do business there? Are their digital contracts upheld by their courts, or is the fact that their signature key is held in escrow in the U.S. going to allow them to claim in Switzerland that someone else could have signed this electronic contract? Or will there be special waivers for people who do business in such countries?

And what about industrial espionage? There is no reason to believe that countries which currently conduct industrial espionage for their important or state-run companies will refrain from doing so on key-escrowed encryption systems. Indeed, since virtually no country is going to allow other countries to oversee its intelligence operations, widespread use of escrowed encryption will probably increase the use of wiretaps.

Even if countries with good civil rights records use key escrow only for the legitimate pursuit of criminals and terrorists, it's certain to be used elsewhere to keep track of dissidents, blackmail political opponents, and so on. Digital communications offer the opportunity to do a much more thorough job of monitoring citizens' actions, opinions, purchases, and associations than is possible in an analog world.

It's not clear how this will affect commercial key escrow, except that 20 years from now, selling Turkey or China a ready-made key-escrow system may look a lot like selling shock batons to South Africa in 1970, or building a chemical plant for Iraq in 1980. Even worse, effortless and untraceable tapping of communications may tempt a number of governments into tracking many of their citizens' communications, even those which haven't generally tried to do so before. And there's no guarantee that liberal democracies will be immune to this temptation.

(3) Each trustee, independently, performs a calculation on its public piece and its private piece to confirm that they are correct. Each trustee stores the private piece somewhere secure and sends the public piece to the KDC.
(4) The KDC performs another calculation on the public pieces and the public key. Assuming that everything is correct, it signs the public key and either sends it back to Alice or posts it in a database somewhere.

If the courts order a wiretap, then each of the trustees surrenders his or her piece to the KDC, and the KDC can reconstruct the private key. Before this surrender, neither the KDC nor any individual trustee can reconstruct the private key; all the trustees are required to reconstruct the key.

Any public-key cryptography algorithm can be made fair in this manner. Some particular algorithms are discussed in Section 23.10. Micali's paper [1084, 1085] discusses ways to combine this with a threshold scheme, so that a subset of the trustees (e.g., three out of five) is required to reconstruct the private key. He also shows how to combine this with oblivious transfer (see Section 5.5) so that the trustees do not know whose private key is being reconstructed.

Fair cryptosystems aren't perfect. A criminal can exploit the system, using a subliminal channel (see Section 4.2) to embed another secret key into his piece. This way, he can communicate securely with someone else using this subliminal key without having to worry about court-authorized wiretapping. Another protocol, called **failsafe key escrowing**, solves this problem [946, 833]. Section 23.10 describes the algorithm and protocol.

The Politics of Key Escrow

Aside from the government's key-escrow plans, several commercial key-escrow proposals are floating around. This leads to the obvious question: What are the advantages of key-escrow for the user?

Well, there really aren't any. The user gains nothing from key escrow that he couldn't provide himself. He can already back up his keys if he wants (see Section 8.8). Key-escrow guarantees that the police can eavesdrop on his conversations or read his data files even though they are encrypted. It guarantees that the NSA can eavesdrop on his international phone calls—without a warrant—even though they are encrypted. Perhaps he will be allowed to use cryptography in countries that now ban it, but that seems to be the only advantage.

Key escrow has considerable disadvantages. The user has to trust the escrow agents' security procedures, as well as the integrity of the people involved. He has to trust the escrow agents not to change their policies, the government not to change its laws, and those with lawful authority to get his keys to do so lawfully and responsibly. Imagine a major terrorist attack in New York; what sorts of limits on the police would be thrown aside in the aftermath?

It is hard to imagine escrowed encryption schemes working as their advocates imagine without some kind of legal pressure. The obvious next step is a ban on the use of non-escrowed encryption. This is probably the only way to make a commercial system pay, and it's certainly the only way to get technologically sophisticated criminals and terrorists to use it. It's not clear how difficult outlawing non-escrowed cryptography will be, or how it will affect cryptography as an academic discipline. How can I research software-oriented cryptography algorithms without having software non-escrowed encryption devices in my possession; will I need a special license?

And there are legal questions. How do escrowed keys affect users' liability, should some encrypted data get out? If the U.S. government is trying to protect the escrow agencies, will there be the implicit assumption that if the secret was compromised by either the user or the escrow agency, then it must have been the user?

4.14 Key Escrow

This excerpt is from Silvio Micali's introduction to the topic [1084]:

> Currently, court-authorized line tapping is an effective method for securing criminals to justice. More importantly, in our opinion, it also prevents the further spread of crime by deterring the use of ordinary communication networks for unlawful purposes. Thus, there is a legitimate concern that widespread use of public-key cryptography may be a big boost for criminal and terrorist organizations. Indeed, many bills propose that a proper governmental agency, under circumstances allowed by law, should be able to obtain the clear text of any communication over a public network. At the present time, this requirement would translate into coercing citizens to either (1) *using weak cryptosystems*—i.e., cryptosystems that the proper authorities (but also everybody else!) could crack with a moderate effort—or (2) *surrendering*, a priori, *their secret key* to the authority. It is not surprising that such alternatives have legitimately alarmed many concerned citizens, generating as reaction the feeling that privacy should come before national security and law enforcement.

Key escrow is the heart of the U.S. government's Clipper program and its Escrowed Encryption Standard. The challenge here is to develop a cryptosystem that both protects individual privacy but at the same time allows for court-authorized wiretaps.

The Escrowed Encryption Standard gets its security from tamperproof hardware. Each encryption chip has a unique ID number and secret key. This key is split into two pieces and stored, along with the ID number, by two different escrow agencies. Every time the chip encrypts a data file, it first encrypts the session key with this unique secret key. Then it transmits this encrypted session key and its ID number over the communications channel. When some law enforcement agency wants to decrypt traffic encrypted with one of these chips, it listens for the ID number, collects the appropriate keys from the escrow agencies, XORs them together, decrypts the session key, and then uses the session key to decrypt the message traffic. There are more complications to make this scheme work in the face of cheaters; see Section 24.16 for details. The same thing can be done in software, using public-key cryptography [77, 1579, 1580, 1581].

Micali calls his idea **fair cryptosystems** [1084, 1085]. (The U.S. government reportedly paid Micali $1, 000, 000 for the use of his patents [1086, 1087] in their Escrowed Encryption Standard; Banker's Trust then bought Micali's patent.) In these cryptosystems, the private key is broken up into pieces and distributed to different authorities. Like a secret sharing scheme, the authorities can get together and reconstruct the private key. However, the pieces have the additional property that they can be individually verified to be correct, without reconstructing the private key.

Alice can create her own private key and give a piece to each of n trustees. None of these trustees can recover Alice's private key. However, each trustee can verify that his piece is a valid piece of the private key; Alice cannot send one of the trustees a random-bit string and hope to get away with it. If the courts authorize a wiretap, the relevant law enforcement authorities can serve a court order on the n trustees to surrender their pieces. With all n pieces, the authorities reconstruct the private key and can wiretap Alice's communications lines. On the other hand, Mallory has to corrupt all n trustees in order to be able to reconstruct Alice's key and violate her privacy.

Here's how the protocol works:

> **(1)** Alice creates her private-key/public-key key pair. She splits the private key into several public pieces and private pieces.
> **(2)** Alice sends a public piece and corresponding private piece to each of the trustees. These messages must be encrypted. She also sends the public key to the KDC.

Cabal Inc. can choose from several solutions. Every member can carry a membership list. This has two problems. One, everyone now has to carry a large database, and two, they have to guard that membership list pretty carefully. Alternatively, a trusted secretary could issue digitally signed ID cards. This has the added advantage of allowing outsiders to verify members (for discounts at the local grocery store, for example), but it requires a trusted secretary. Nobody at Cabal, Inc. can be trusted to that degree.

A novel solution is to use something called a **one-way accumulator** [116]. This is sort of like a one-way hash function, except that it is commutative. That is, it is possible to hash the database of members in any order and get the same value. Moreover, it is possible to add members into the hash and get a new hash, again without regard to order.

So, here's what Alice does. She calculates the accumulation of every member's name other than herself. Then she saves that single value along with her own name. Bob, and every other member, does the same. Now, when Alice and Bob meet in the dimly lit restaurant, they simply trade accumulations and names with each other. Alice confirms that Bob's name added to his accumulation is equal to Alice's name added to her accumulation. Bob does the same. Now they both know that the other is a member. And at the same time, neither can figure out the identities of any other member.

Even better, nonmembers can be given the accumulation of everybody. Now Alice can verify her membership to a nonmember (for membership discounts at their local counterspy shop, perhaps) without the nonmember being able to figure out the entire membership list.

New members can be added just by sending around the new names. Unfortunately, the only way to delete a member is to send everyone a new list and have them recompute their accumulations. But Cabal, Inc. only has to do that if a member resigns; dead members can remain on the list. (Oddly enough, this has never been a problem.)

This is a clever idea, and has applications whenever you want the same effect as digital signatures without a centralized signer.

4.13 All-or-Nothing Disclosure of Secrets

Imagine that Alice is a former agent of the former Soviet Union, now unemployed. In order to make money, Alice sells secrets. Anyone who is willing to pay the price can buy a secret. She even has a catalog. All her secrets are listed by number, with tantalizing titles: "Where is Jimmy Hoffa?", "Who is secretly controlling the Trilateral Commission?", "Why does Boris Yeltsin always look like he swallowed a live frog?", and so on.

Alice won't give away two secrets for the price of one or even partial information about any of the secrets. Bob, a potential buyer, doesn't want to pay for random secrets. He also doesn't want to tell Alice which secrets he wants. It's none of Alice's business, and besides, Alice could then add "what secrets Bob is interested in" to her catalog.

A poker protocol won't work in this case, because at the end of the protocol Alice and Bob have to reveal their hands to each other. There are also tricks Bob can do to learn more than one secret.

The solution is called **all-or-nothing disclosure of secrets (ANDOS)** [246] because, as soon as Bob has gained any information whatsoever about one of Alice's secrets, he has wasted his chance to learn anything about any of the other secrets.

There are several ANDOS protocols in the cryptographic literature. Some of them are discussed in Section 23.9.

Other research on poker protocols can be found in [573, 1634, 389]. A complicated protocol that allows players to not reveal their hands can be found in [390]. Don Coppersmith discusses two ways to cheat at mental poker using the RSA algorithm [370].

Anonymous Key Distribution

While it is unlikely that anyone is going to use this protocol to play poker via modem, Charles Pfleeger discusses a situation in which this type of protocol would come in handy [1244].

Consider the problem of key distribution. If we assume that people cannot generate their own keys (they might have to be of a certain form, or have to be signed by some organization, or something similar), we have to set up a Key Distribution Center to generate and distribute keys. The problem is that we have to figure out some way of distributing keys such that no one, including the server, can figure out who got which key.

This protocol solves the problem:

(1) Alice generates a public-key/private-key key pair. For this protocol, she keeps both keys secret.
(2) The KDC generates a continuous stream of keys.
(3) The KDC encrypts the keys, one by one, with its own public key.
(4) The KDC transmits the encrypted keys, one by one, onto the network.
(5) Alice chooses a key at random.
(6) Alice encrypts the chosen key with her public key.
(7) Alice waits a while (long enough so the server has no idea which key she has chosen) and sends the double-encrypted key back to the KDC.
(8) The KDC decrypts the double-encrypted key with its private key, leaving a key encrypted with Alice's public key.
(9) The server sends the encrypted key back to Alice.
(10) Alice decrypts the key with her private key.

Eve, sitting in the middle of this protocol, has no idea what key Alice chose. She sees a continuous stream of keys go by in step (4). When Alice sends the key back to the server in step (7), it is encrypted with her public key, which is also secret during this protocol. Eve has no way of correlating it with the stream of keys. When the server sends the key back to Alice in step (9), it is also encrypted with Alice's public key. Only when Alice decrypts the key in step (10) is the key revealed.

If you use RSA, this protocol leaks information at the rate of one bit per message. It's the quadratic residues again. If you're going to distribute keys in this manner, make sure this leakage isn't enough to matter. Also, the stream of keys from the KDC must be great enough to preclude a brute-force attack. Of course, if Alice can't trust the KDC, then she shouldn't be getting keys from it. A malicious KDC could presumably keep records of every key it generates. Then, it could search them all to determine which is Alice's.

This protocol also assumes that Alice is going to act fairly. There are things she can do, using RSA, to get more information than she might otherwise. This is not a problem in our scenario, but can be in other circumstances.

4.12 One-Way Accumulators

Alice is a member of Cabal, Inc. Occasionally she has to meet with other members in dimly lit restaurants and whisper secrets back and forth. The problem is that the restaurants are so dimly lit that she has trouble knowing if the person across the table from her is also a member.

$D_C(E_C(M_n)) = M_n$

(8) Carol chooses five more messages at random from the remaining 42. She sends them to Alice.

$E_A(M_n)$

(9) Alice decrypts the messages with her private key to reveal her hand.

$D_A(E_A(M_n)) = M_n$

(10) At the end of the game Alice, Bob, and Carol all reveal their hands and all of their keys so that everyone can make sure that no one has cheated.

Additional cards can be dealt in the same manner. If Bob or Carol wants a card, either one can take the encrypted deck and go through the protocol with Alice. If Alice wants a card, whoever currently has the deck sends her a random card.

Ideally, step (10) would not be necessary. All players shouldn't be required to reveal their hands at the end of the protocol; only those who haven't folded. Since step (10) is part of the protocol designed only to catch cheaters, perhaps there are improvements.

In poker, one is only interested in whether the winner cheated. Everyone else can cheat as much as they want, as long as they still lose. (Actually, this is not really true. Someone can, while losing, collect data on another player's poker style.) So, let's look at cases in which different players win.

If Alice wins, she reveals her hand and her keys. Bob can use Alice's private key to confirm that Alice performed step (2) correctly—that each of the 52 messages corresponded to a different card. Carol can confirm that Alice is not lying about her hand by encrypting the cards with Alice's public key and verifying that they are the same as the encrypted messages she sent to her in step (8).

If either Bob or Carol wins, the winner reveals his hand and keys. Alice can confirm that the cards are legitimate by checking her random strings. She can also confirm that the cards are the ones dealt by encrypting the cards with the winner's public key and verifying that they are the same as the encrypted messages she received in step (3) or (5).

This protocol isn't secure against collusion among malicious players. Alice and another player can effectively gang up on the third and together swindle that player out of everything without raising suspicion. Therefore, it is important to check all the keys and random strings every time the players reveal their hands. And if you're sitting around the virtual table with two people who never reveal their hands whenever one of them is the dealer (Alice, in the previous protocol), stop playing.

Understand that while this is all interesting theory, actually implementing it on a computer is an arduous task. A Sparc implementation with three players on separate workstations takes eight hours to shuffle a deck of cards, let alone play an actual game [513].

Attacks against Poker Protocols

Cryptographers have shown that a small amount of information is leaked by these poker protocols if the RSA public-key algorithm is used [453, 573]. Specifically, if the binary representation of the card is a quadratic residue (see Section 11.3), then the encryption of the card is also a quadratic residue. This property can be used to "mark" some cards—all the aces, for example. This does not reveal much about the hands, but in a game such as poker even a tiny bit of information can be an advantage in the long run.

Shafi Goldwasser and Silvio Micali [624] developed a two-player mental-poker protocol that fixes this problem, although its complexity makes it far more theoretical than practical. A general *n*-player poker protocol that eliminates the problem of information leakage was developed in [389].

It is interesting to note that in all these protocols, Alice and Bob don't learn the result of the coin flip at the same time. Each protocol has a point where one of the parties (Alice in the first two protocols and Bob in the last one) knows the result of the coin flip but cannot change it. That party can, however, delay disclosing the result to the other party. This is known as **flipping coins into a well**. Imagine a well. Alice is next to the well and Bob is far away. Bob throws the coin and it lands in the well. Alice can now look into the well and see the result, but she cannot reach down to change it. Bob cannot see the result until Alice lets him come close enough to look.

Key Generation Using Coin Flipping

A real application for this protocol is session-key generation. Coin-flipping protocols allow Alice and Bob to generate a random session key such that neither can influence what the session key will be. And assuming that Alice and Bob encrypt their exchanges, this key generation is secure from eavesdropping as well.

4.11 Mental Poker

A protocol similar to the public-key fair coin flip protocol allows Alice and Bob to play poker with each other via electronic mail. Instead of Alice making and encrypting two messages, one for heads and one for tails, she makes 52 messages, M_1, M_2,..., M_{52}, one for each card in the deck. Bob chooses five messages at random, encrypts them with his public key, and then sends them back to Alice. Alice decrypts the messages and sends them back to Bob, who decrypts them to determine his hand. He then chooses five more messages at random and sends them back to Alice as he received them; she decrypts these and they become her hand. During the game, additional cards can be dealt to either player by repeating the procedure. At the end of the game, Alice and Bob both reveal their cards and key pairs so that each can be assured that the other did not cheat.

Mental Poker with Three Players

Poker is more fun with more players. The basic mental poker protocol can easily be extended to three or more players. In this case, too, the cryptographic algorithm must be commutative.

(1) Alice, Bob, and Carol each generate a public-key/private-key key pair.
(2) Alice generates 52 messages, one for each card in the deck. These messages should contain some unique random string, so that she can verify their authenticity later in the protocol. Alice encrypts all the messages with her public key and sends them to Bob.

$$E_A(M_n)$$

(3) Bob, who cannot read any of the messages, chooses five at random. He encrypts them with his public key and sends them back to Alice.

$$E_B(E_A(M_n))$$

(4) Bob sends the other 47 messages to Carol.

$$E_A(M_n)$$

(5) Carol, who cannot read any of the messages, chooses five at random. She encrypts them with her public key and sends them to Alice.

$$E_C(E_A(M_n))$$

(6) Alice, who cannot read any of the messages sent back to her, decrypts them with her private key and then sends them back to Bob or Carol (depending on where they came from).

$$D_A(E_B(E_A(M_n))) = E_B(M_n)$$
$$D_A(E_C(E_A(M_n))) = E_C(M_n)$$

(7) Bob and Carol decrypt the messages with their keys to reveal their hands.

$$D_B(E_B(M_n)) = M_n$$

requirement is that the algorithm commute. That is:

$$D_{K1}(E_{K2}(E_{K1}(M))) = E_{K2}(M)$$

In general, this property is not true for symmetric algorithms, but it is true for some public-key algorithms (RSA with identical moduli, for example). This is the protocol:

(1) Alice and Bob each generate a public-key/private-key key pair.
(2) Alice generates two messages, one indicating heads and the other indicating tails. These messages should contain some unique random string, so that she can verify their authenticity later in the protocol. Alice encrypts both messages with her public key and sends them to Bob in a random order.

$E_A(M_1)$, $E_A(M_2)$

(3) Bob, who cannot read either message, chooses one at random. (He can sing "eeny meeny miney moe, " engage a malicious computer intent on subverting the protocol, or consult the *I Ching*—it doesn't matter.) He encrypts it with his public key and sends it back to Alice.

$E_B(E_A(M))$

M is either M_1 or M_2.

(4) Alice, who cannot read the message sent back to her, decrypts it with her private key and then sends it back to Bob.

$D_A(E_B(E_A(M))) = E_B(M_1)$ if $M = M_1$, or
$E_B(M_2)$ if $M = M_2$

(5) Bob decrypts the message with his private key to reveal the result of the coin flip. He sends the decrypted message to Alice.

$D_B(E_B(M_1)) = M_1$ or $D_B(E_B(M_2)) = M_2$

(6) Alice reads the result of the coin flip and verifies that the random string is correct.
(7) Both Alice and Bob reveal their key pairs so that both can verify that the other did not cheat.

This protocol is self-enforcing. Either party can immediately detect cheating by the other, and no trusted third party is required to participate in either the actual protocol or any adjudication after the protocol has been completed. To see how this works, let's try to cheat.

If Alice wanted to cheat and force heads, she has three potential ways of affecting the outcome. First, she could encrypt two "heads" messages in step (2). Bob would discover this when Alice revealed her keys at step (7). Second, she could use some other key to decrypt the message in step (4). This would result in gibberish, which Bob would discover in step (5). Third, she could lie about the validity of the message in step (6). Bob would also discover this in step (7), when Alice could not prove that the message was not valid. Of course, Alice could refuse to participate in the protocol at any step, at which point Alice's attempted deception would be obvious to Bob.

If Bob wanted to cheat and force "tails, " his options are just as poor. He could incorrectly encrypt a message at step (3), but Alice would discover this when she looked at the final message at step (6). He could improperly perform step (5), but this would also result in gibberish, which Alice would discover at step (6). He could claim that he could not properly perform step (5) because of some cheating on the part of Alice, but this form of cheating would be discovered at step (7). Finally, he could send a "tails" message to Alice at step (5), regardless of the message he decrypted, but Alice would immediately be able to check the message for authenticity at step (6).

Flipping Coins into a Well

random bit, *x*, exclusive-*OR*ed with any independently distributed bit, *y*, will yield a truly random bit, Alice's protocol did not ensure that the two bits were distributed independently. In fact, it is not hard to verify that no mental protocol can allow two infinitely powerful parties to flip a fair coin. Alice and Bob were in trouble until they received a letter from an obscure graduate student in cryptography. The information in the letter was too theoretical to be of any earthly use to anyone, but the envelope the letter came in was extremely handy.

The next time Alice and Bob wished to flip a coin, they played a modified version of the original protocol. First, Bob decided on a bit, but instead of announcing it immediately, he wrote it down on a piece of paper and placed the paper in the envelope. Next, Alice announced her bit. Finally, Alice and Bob took Bob's bit out of the envelope and computed the random bit. This bit was indeed truly random whenever at least one of them played honestly. Alice and Bob had a working protocol, the cryptographer's dream of social relevance was fulfilled, and they all lived happily ever after.

Those envelopes sound a lot like bit-commitment blobs. When Manuel Blum introduced the problem of flipping a fair coin over a modem [194], he solved it using a bit-commitment protocol:

(1) Alice commits to a random bit, using any of the bit-commitment schemes listed in Section 4.9.
(2) Bob tries to guess the bit.
(3) Alice reveals the bit to Bob. Bob wins the flip if he correctly guessed the bit.

In general, we need a protocol with these properties:

— Alice must flip the coin before Bob guesses.
— Alice must not be able to re-flip the coin after hearing Bob's guess.
— Bob must not be able to know how the coin landed before making his guess.

There are several ways in which we can do this.

Coin Flipping Using One-Way Functions

If Alice and Bob can agree on a one-way function, this protocol is simple:

(1) Alice chooses a random number, *x*. She computes $y = f(x)$, where $f(x)$ is the one-way function.
(2) Alice sends *y* to Bob.
(3) Bob guesses whether *x* is even or odd and sends his guess to Alice.
(4) If Bob's guess is correct, the result of the coin flip is heads. If Bob's guess is incorrect, the result of the coin flip is tails. Alice announces the result of the coin flip and sends *x* to Bob.
(5) Bob confirms that $y = f(x)$.

The security of this protocol rests in the one-way function. If Alice can find *x* and *x´*, such that *x* is even and *x´* is odd, and $y = f(x) = f(x´)$, then she can cheat Bob every time. The least significant bit of $f(x)$ must also be uncorrelated with *x*. If not, Bob can cheat Alice at least some of the time. For example, if $f(x)$ produces even numbers 75 percent of the time if *x* is even, Bob has an advantage. (Sometimes the least significant bit is not the best one to use in this application, because it can be easier to compute.)

Coin Flipping Using Public-Key Cryptography

This protocol works with either public-key cryptography or symmetric cryptography. The only

When it comes time for Alice to reveal her bit, the protocol continues:

(3) Alice sends Bob her random seed.
(4) Bob completes step (2) to confirm that Alice was acting fairly.

If Bob's random-bit string is long enough, and the pseudo-random-bit generator is unpredictable, then there is no practical way Alice can cheat.

Blobs

These strings that Alice sends to Bob to commit to a bit are sometimes called **blobs**. A blob is a sequence of bits, although there is no reason in the protocols why it has to be. As Gilles Brassard said, "They could be made out of fairy dust if this were useful" [236]. Blobs have these four properties:

1. Alice can commit to blobs. By committing to a blob, she is committing to a bit.
2. Alice can open any blob she has committed to. When she opens a blob, she can convince Bob of the value of the bit she committed to when she committed to the blob. Thus, she cannot choose to open any blob as either a zero or a one.
3. Bob cannot learn how Alice is able to open any unopened blob she has committed to. This is true even after Alice has opened other blobs.
4. Blobs do not carry any information other than the bit Alice committed to. The blobs themselves, as well as the process by which Alice commits to and opens them, are uncorrelated to anything else that Alice might wish to keep secret from Bob.

4.10 Fair Coin Flips

It's story time with Joe Kilian [831]:

> Alice and Bob wanted to flip a fair coin, but had no physical coin to flip. Alice offered a simple way of flipping a fair coin mentally.
>
> "First, you think up a random bit, then I'll think up a random bit. We'll then exclusive-or the two bits together, " she suggested.
>
> "But what if one of us doesn't flip a coin at random?" Bob asked.
>
> "It doesn't matter. As long as one of the bits is truly random, the exclusive-or of the bits should be truly random, " Alice replied, and after a moment's reflection, Bob agreed.
>
> A short while later, Alice and Bob happened upon a book on artificial intelligence, lying abandoned by the roadside. A good citizen, Alice said, "One of us must pick this book up and find a suitable waste receptacle." Bob agreed, and suggested they use their coin-flipping protocol to determine who would have to throw the book away.
>
> "If the final bit is a 0, then you will pick the book up, and if it is a 1, then I will, " said Alice. "What is your bit?"
>
> Bob replied, "1."
>
> "Why, so is mine, " said Alice, slyly, "I guess this isn't your lucky day."
>
> Needless to say, this coin-flipping protocol had a serious bug. While it is true that a truly

handed Bob with a variety of keys until she found one that gave her a bit other than the one she committed to. Since the bit has only two possible values, she is certain to find one after only a few tries. Bob's random string prevents her from using this attack; she has to find a new message that not only has her bit inverted, but also has Bob's random string exactly reproduced. If the encryption algorithm is good, the chance of her finding this is minuscule. Alice cannot change her bit after she commits to it.

Bit Commitment Using One-Way Functions

This protocol uses one-way functions:

(1) Alice generates two random-bit strings, R_1 and R_2.

R_1,R_2

(2) Alice creates a message consisting of her random strings and the bit she wishes to commit to (it can actually be several bits).

(R_1,R_2,b)

(3) Alice computes the one-way function on the message and sends the result, as well as one of the random strings, to Bob.

$H(R_1,R_2,b),R_1$

This transmission from Alice is evidence of commitment. Alice's one-way function in step (3) prevents Bob from inverting the function and determining the bit.

When it comes time for Alice to reveal her bit, the protocol continues:

(4) Alice sends Bob the original message.

(R_1,R_2,b)

(5) Bob computes the one-way function on the message and compares it and R_1, with the value and random string he received in step (3). If they match, the bit is valid.

The benefit of this protocol over the previous one is that Bob does not have to send any messages. Alice sends Bob one message to commit to a bit and another message to reveal the bit.

Bob's random string isn't required because the result of Alice's commitment is a message operated on by a one-way function. Alice cannot cheat and find another message (R_1,R_2',b'), such that $H(R_1,R_2',b') = H(R_1,R_2,b)$. By sending Bob R_1 she is committing to the value of b. If Alice didn't keep R_2 secret, then Bob could compute both $H(R_1,R_2,b)$ and $H(R_1,R_2,b')$ and see which was equal to what he received from Alice.

Bit Commitment Using Pseudo-Random-Sequence Generators

This protocol is even easier [1137]:

(1) Bob generates a random-bit string and sends it to Alice.

R_B

(2) Alice generates a random seed for a pseudo-random-bit generator. Then, for every bit in Bob's random-bit string, she sends Bob either:

(a) the output of the generator if Bob's bit is 0, or

(b) the XOR of output of the generator and her bit, if Bob's bit is 1.

4.9 Bit Commitment

The Amazing Alice, magician extraordinaire, will now perform a mystifying feat of mental prowess. She will guess the card Bob will choose before he chooses it! Watch as Alice writes her prediction on a piece of paper. Marvel as Alice puts that piece of paper in an envelope and seals it shut. Thrill as Alice hands that sealed envelope to a random member of the audience. "Pick a card, Bob, any card." He looks at it and shows it to Alice and the audience. It's the seven of diamonds. Alice now takes the envelope back from the audience. She rips it open. The prediction, written before Bob chose his card, says "seven of diamonds"! Applause.

To make this work, Alice had to switch envelopes at the end of the trick. However, cryptographic protocols can provide a method immune from any sleight of hand. Why is this useful? Here's a more mundane story:

Stockbroker Alice wants to convince investor Bob that her method of picking winning stocks is sound.

> Bob: "Pick five stocks for me. If they are all winners, I'll give you my business."
>
> Alice: "If I pick five stocks for you, you could invest in them without paying me. Why don't I show you the stocks I picked last month?"
>
> Bob: "How do I know you didn't change last month's picks after you knew their outcome? If you tell me your picks now, I'll know that you can't change them. I won't invest in those stocks until after I've purchased your method. Trust me."
>
> Alice: "I'd rather show you my picks from last month. I didn't change them. Trust me."

Alice wants to commit to a prediction (i.e., a bit or series of bits) but does not want to reveal her prediction until sometime later. Bob, on the other hand, wants to make sure that Alice cannot change her mind after she has committed to her prediction.

Bit Commitment Using Symmetric Cryptography

This bit-commitment protocol uses symmetric cryptography:

(1) Bob generates a random-bit string, *R*, and sends it to Alice.
R
(2) Alice creates a message consisting of the bit she wishes to commit to, *b* (it can actually be several bits), and Bob's random string. She encrypts it with some random key, *K*, and sends the result back to Bob.
$E_K(R,b)$

That is the commitment portion of the protocol. Bob cannot decrypt the message, so he does not know what the bit is.

When it comes time for Alice to reveal her bit, the protocol continues:

(3) Alice sends Bob the key.
(4) Bob decrypts the message to reveal the bit. He checks his random string to verify the bit's validity.

If the message did not contain Bob's random string, Alice could secretly decrypt the message she

keys each member uses.

Chaum [330] lists a number of other protocols, some in which Trent is unable to fake signatures and others in which Trent is not even required. Another protocol [348] not only hides the identity of the signer, but also allows new members to join the group. Yet another protocol is [1230].

4.7 Fail-Stop Digital Signatures

Let's say Eve is a very powerful adversary. She has vast computer networks and rooms full of Cray computers—orders of magnitude more computing power than Alice. All of these computers chug away, day and night, trying to break Alice's private key. Finally—success. Eve can now impersonate Alice, forging her signature on documents at will.

Fail-stop digital signatures, introduced by Birgit Pfitzmann and Michael Waidner [1240], prevent this kind of cheating. If Eve forges Alice's signatures after a brute-force attack, then Alice can prove they are forgeries. If Alice signs a document and then disavows the signature, claiming forgery, a court can verify that it is not a forgery.

The basic idea behind fail-stop signatures is that for every possible public key, many possible private keys work with it. Each of these private keys yields many different possible signatures. However, Alice has only one private key and can compute just one signature. Alice doesn't know any of the other private keys.

Eve wants to break Alice's private key. (Eve could also be Alice, trying to compute a second private key for herself.) She collects signed messages and, using her array of Cray computers, tries to recover Alice's private key. Even if she manages to recover a valid private key, there are so many possible private keys that it is far more likely that she has a different one. The probability of Eve's recovering the proper private key can be made so small as to be negligible.

Now, when Eve forges a signed document using the private key she generated, it will have a different signature than if Alice signs the document herself. When Alice is hauled off to court, she can produce two different signatures for the same message and public key (corresponding to her private key and to the private key Eve created) to prove forgery. On the other hand, if Alice cannot produce the two different signatures, there is no forgery and Alice is still bound by her signature.

This signature scheme protects against Eve breaking Alice's signature scheme by sheer computational power. It does nothing against Mallory's much more likely attack of breaking into Alice's house and stealing her private key or Alice's attack of signing a document and then conveniently losing her private key. To protect against the former, Alice should buy herself a good guard dog; that kind of thing is beyond the scope of cryptography.

Additional theory and applications of fail-stop signatures can be found in [1239, 1241, 730, 731].

4.8 Computing with Encrypted Data

Alice wants to know the solution to some function $f(x)$, for some particular value of x. Unfortunately, her computer is broken. Bob is willing to compute $f(x)$ for her, but Alice isn't keen on letting Bob know her x. How can Alice let Bob compute $f(x)$ for her without telling him x?

This is the general problem of **computing with encrypted data**, also called **hiding information from an oracle**. (Bob is the oracle; he answers questions.) There are ways to do this for certain functions; they are discussed in Section 23.6.

— ***Distinguishability***. Proxy signatures are distinguishable from normal signatures by anyone.
— ***Unforgeability***. Only the original signer and the designated proxy signer can create a valid proxy signature.
— ***Proxy signer's deviation***. A proxy signer cannot create a valid proxy signature not detected as a proxy signature.
— ***Verifiability***. From a proxy signature, a verifier can be convinced of the original signer's agreement on the signed message.
— ***Identifiability***. An original signer can determine the proxy signer's identity from a proxy signature.
— ***Undeniability***. A proxy signer cannot disavow an accepted proxy signature he created.

In some cases, a stronger form of identifiability is required—that anyone can determine the proxy signer's identity from the proxy signature. Proxy signature schemes, based on different digital signature schemes, are in [1001].

4.6 Group Signatures

David Chaum introduces this problem in [330]:

> A company has several computers, each connected to the local network. Each department of that company has its own printer (also connected to the network) and only persons of that department are allowed to use their department's printer. Before printing, therefore, the printer must be convinced that the user is working in that department. At the same time, the company wants privacy; the user's name may not be revealed. If, however, someone discovers at the end of the day that a printer has been used too often, the director must be able to discover who misused that printer, and send him a bill.

The solution to this problem is called a **group signature**. Group signatures have the following properties:

— Only members of the group can sign messages.
— The receiver of the signature can verify that it is a valid signature from the group.
— The receiver of the signature cannot determine which member of the group is the signer.
— In the case of a dispute, the signature can be "opened" to reveal the identity of the signer.

Group Signatures with a Trusted Arbitrator

This protocol uses a trusted arbitrator:

(1) Trent generates a large pile of public-key/private-key key pairs and gives every member of the group a different list of unique private keys. No keys on any list are identical. (If there are *n* members of the group, and each member gets *m* key pairs, then there are *n***m* total key pairs.)
(2) Trent publishes the master list of all public keys for the group, in random order. Trent keeps a secret record of which keys belong to whom.
(3) When group members want to sign a document, he chooses a key at random from his personal list.
(4) When someone wants to verify that a signature belongs to the group, he looks on the master list for the corresponding public key and verifies the signature.
(5) In the event of a dispute, Trent knows which public key corresponds to which group member.

The problem with this protocol is that it requires a trusted party. Trent knows everyone's private keys and can forge signatures. Also, *m* must be long enough to preclude attempts to analyze which

Inc., and sends incriminating documents to a newspaper using an undeniable signature protocol. Alice can verify her signature to the newspaper reporter, but not to anyone else. However, CEO Bob suspects that Alice is the source of the documents. He demands that Alice run the disavowal protocol to clear her name, and Alice refuses. Bob maintains that the only reason Alice has to refuse is that she is guilty, and fires her.

Entrusted undeniable signatures are like undeniable signatures, except that the disavowal protocol can only be run by Trent. Bob cannot demand that Alice run the disavowal protocol; only Trent can. And if Trent is the court system, then he will only run the protocol to resolve a formal dispute.

4.4 Designated Confirmer Signatures

The Alice Software Company is doing a booming business selling DEW—so good, in fact, that Alice is spending more time verifying undeniable signatures than writing new features.

Alice would like a way to designate one particular person in the company to be in charge of signature verification for the whole company. Alice, or any other programmer, would be able to sign documents with an undeniable protocol. But the verifications would all be handled by Carol.

As it turns out, this is possible with **designated confirmer signatures** [333, 1213]. Alice can sign a document such that Bob is convinced the signature is valid, but he cannot convince a third party; at the same time Alice can designate Carol as the future confirmer of her signature. Alice doesn't even need to ask Carol's permission beforehand; she just has to use Carol's public key. And Carol can still verify Alice's signature if Alice is out of town, has left the company, or just upped and died.

Designated confirmer signatures are kind of a compromise between normal digital signatures and undeniable signatures. There are certainly instances where Alice might want to limit who can verify her signature. On the other hand, giving Alice complete control undermines the enforceability of signatures: Alice might refuse to cooperate in either confirming or denying, she might claim the loss of keys for confirming or denying, or she might just be unavailable. Designated confirmer signatures can give Alice the protection of an undeniable signature while not letting her abuse that protection. Alice might even prefer it that way: Designated confirmer signatures can help prevent false applications, protect her if she actually does lose her key, and step in if she is on vacation, in the hospital, or even dead.

This idea has all sorts of possible applications. Carol can set herself up as a notary public. She can publish her public key in some directory somewhere, and people can designate her as a confirmer for their signatures. She can charge a small fee for confirming signatures for the masses and make a nice living.

Carol can be a copyright office, a government agency, or a host of other things. This protocol allows organizations to separate the people who sign documents from the people who help verify signatures.

4.5 Proxy Signatures

Designated confirmer signatures allows a signer to designate someone else to verify his signature. Alice, for instance, needs to go on a business trip to someplace which doesn't have very good computer network access—to the jungles of Africa, for example. Or maybe she is incapacitated after major surgery. She expects to receive some important e-mail, and has instructed her secretary Bob to respond accordingly. How can Alice give Bob the power to sign messages for her, without giving him her private key?

Proxy signatures is a solution [1001]. Alice can give Bob a proxy, such that the following properties hold:

The Alice Software Company distributes DEW (Do-Everything-Word). To ensure that their software is virus-free, they include a digital signature with each copy. However, they want only legitimate buyers of the software, not software pirates, to be able to verify the signature. At the same time, if copies of DEW are found to contain a virus, the Alice Software Company should be unable to deny a valid signature.

Undeniable signatures [343, 327] are suited to these sorts of tasks. Like a normal digital signature, an undeniable signature depends on the signed document and the signer's private key. But unlike normal digital signatures, an undeniable signature cannot be verified without the signer's consent. Although a better name for these signatures might be something like "nontransferable signatures, " the name comes from the fact that if Alice is forced to either acknowledge or deny a signature—perhaps in court—she cannot falsely deny her real signature.

The mathematics are complicated, but the basic idea is simple:

(1) Alice presents Bob with a signature.
(2) Bob generates a random number and sends it to Alice.
(3) Alice does a calculation using the random number and her private key and sends Bob the result. Alice could only do this calculation if the signature is valid.
(4) Bob confirms this.

There is also an additional protocol so that Alice can prove that she did not sign a document, and cannot falsely deny a signature.

Bob can't turn around and convince Carol that Alice's signature is valid, because Carol doesn't know that Bob's numbers are random. He could have easily worked the protocol backwards on paper, without any help from Alice, and then shown Carol the result. Carol can be convinced that Alice's signature is valid only if she completes the protocol with Alice herself. This might not make much sense now, but it will once you see the mathematics in Section 23.4.

This solution isn't perfect. Yvo Desmedt and Moti Yung show that it is possible, in some applications, for Bob to convince Carol that Alice's signature is valid [489].

For instance, Bob buys a legal copy of DEW. He can validate the signature on the software package whenever he wants. Then, Bob convinces Carol that he's a salesman from the Alice Software Company. He sells her a pirated copy of DEW. When Carol tries to validate the signature with Bob, he simultaneously validates the signature with Alice. When Carol sends him the random number, he then sends it on to Alice. When Alice replies, he then sends the reply on to Carol. Carol is convinced that she is a legitimate buyer of the software, even though she isn't. This attack is an instance of the chess grandmaster problem and is discussed in detail in Section 5.2.

Even so, undeniable signatures have a lot of applications; in many instances Alice doesn't want anyone to be able to verify her signature. She might not want personal correspondence to be verifiable by the press, be shown and verified out of context, or even to be verified after things have changed. If she signs a piece of information she sold, she won't want someone who hasn't paid for the information to be able to verify its authenticity. Controlling who verifies her signature is a way for Alice to protect her personal privacy.

A variant of undeniable signatures separates the relation between signer and message from the relation between signer and signature [910]. In one signature scheme, anyone can verify that the signer actually created the signature, but the cooperation of the signer is required to verify that the signature is valid for the message.

A related notion is an **entrusted undeniable signature** [1229]. Imagine that Alice works for Toxins,

(6) Bob ignores the innocuous message and, using the secret key he shares with Alice, extracts the subliminal message.

What about cheating? Walter doesn't trust anyone and no one trusts him. He can always prevent communication, but he has no way of introducing phony messages. Since he can't generate any valid signatures, Bob will detect his attempt in step (5). And since he does not know the shared key, he can't read the subliminal messages. Even more important, he has no idea that the subliminal messages are there. Signed messages using a digital signature algorithm look no different from signed messages with subliminal messages embedded in the signature.

Cheating between Alice and Bob is more problematic. In some implementations of a subliminal channel, the secret information Bob needs to read the subliminal message is the same information Alice needs to sign the innocuous message. If this is the case, Bob can impersonate Alice. He can sign messages purporting to come from her, and there is nothing Alice can do about it. If she is to send him subliminal messages, she has to trust him not to abuse her private key.

Other subliminal channel implementations don't have this problem. A secret key shared by Alice and Bob allows Alice to send Bob subliminal messages, but it is not the same as Alice's private key and does not allow Bob to sign messages. Alice need not trust Bob not to abuse her private key.

Applications of Subliminal Channel

The most obvious application of the subliminal channel is in a spy network. If everyone sends and receives signed messages, spies will not be noticed sending subliminal messages in signed documents. Of course, the enemy's spies can do the same thing.

Using a subliminal channel, Alice could safely sign a document under threat. She would, when signing the document, imbed the subliminal message, saying, "I am being coerced." Other applications are more subtle. A company can sign documents and embed subliminal messages, allowing them to be tracked throughout the documents' lifespans. The government can "mark" digital cash. A malicious signature program can leak secret information in its signatures. The possibilities are endless.

Subliminal-Free Signatures

Alice and Bob are sending signed messages to each other, negotiating the terms of a contract. They use a digital signature protocol. However, this contract negotiation has been set up as a cover for Alice's and Bob's spying activities. When they use the digital signature algorithm, they don't care about the messages they are signing. They are using a subliminal channel in the signatures to send secret information to each other. The counterespionage service, however, doesn't know that the contract negotiations and the use of signed messages are just cover-ups. This concern has led people to create **subliminal-free signature schemes**. These digital signature schemes cannot be modified to contain a subliminal channel. See [480, 481] for details.

4.3 Undeniable Digital Signatures

Normal digital signatures can be copied exactly. Sometimes this property is useful, as in the dissemination of public announcements. Other times it could be a problem. Imagine a digitally signed personal or business letter. If many copies of that document were floating around, each of which could be verified by anyone, this could lead to embarrassment or blackmail. The best solution is a digital signature that can be proven valid, but that the recipient cannot show to a third party without the signer's consent.

protocols. In their first version, clients send “certify” requests to a central coordinating server. Following Merkle’s technique of using hash functions to build trees [1066], the server builds a tree of hash values whose leaves are all the requests received during a given second, and sends back to each requester the list of hash values hanging off the path from its leaf to the root of the tree. The client software stores this locally, and can issue a Digital Notary “certificate” for any file that has been certified. The sequence of roots of these trees comprises the “Universal Validation Record” that will be available electronically at multiple repository sites (and also published on CD-ROM). The client software also includes a “validate” function, allowing the user to test whether a file has been certified in exactly its current form (by querying a repository for the appropriate tree root and comparing it against a hash value appropriately recomputed from the file and its certificate). For information contact Surety Technologies, 1 Main St., Chatham, NJ, 07928; (201) 701-0600; Fax: (201) 701-0601.

4.2 Subliminal Channel

Alice and Bob have been arrested and are going to prison. He’s going to the men’s prison and she’s going to the women’s prison. Walter, the warden, is willing to let Alice and Bob exchange messages, but he won’t allow them to be encrypted. Walter expects them to coordinate an escape plan, so he wants to be able to read everything they say.

Walter also hopes to deceive either Alice or Bob. He wants one of them to accept a fraudulent message as a genuine message from the other. Alice and Bob go along with this risk of deception, otherwise they cannot communicate at all, and they have to coordinate their plans. To do this they have to deceive the warden and find a way of communicating secretly. They have to set up a **subliminal channel**, a covert communications channel between them in full view of Walter, even though the messages themselves contain no secret information. Through the exchange of perfectly innocuous signed messages they will pass secret information back and forth and fool Walter, even though Walter is watching all the communications.

An easy subliminal channel might be the number of words in a sentence. An odd number of words in a sentence might correspond to “1, ” while an even number of words might correspond to “0.” So, while you read this seemingly innocent paragraph, I have sent my operatives in the field the message “101.” The problem with this technique is that it is mere steganography (see Section 1.2); there is no key and security depends on the secrecy of the algorithm.

Gustavus Simmons invented the concept of a subliminal channel in a conventional digital signature algorithm [1458, 1473]. Since the subliminal messages are hidden in what looks like normal digital signatures, this is a form of obfuscation. Walter sees signed innocuous messages pass back and forth, but he completely misses the information being sent over the subliminal channel. In fact, the subliminal-channel signature algorithm is indistinguishable from a normal signature algorithm, at least to Walter. Walter not only cannot read the subliminal message, but he also has no idea that one is even present.

In general the protocol looks like this:

(1) Alice generates an innocuous message, pretty much at random.
(2) Using a secret key shared with Bob, Alice signs the innocuous message in such a way that she hides her subliminal message in the signature. (This is the meat of the subliminal channel protocol; see Section 23.3.)
(3) Alice sends this signed message to Bob via Walter.
(4) Walter reads the innocuous message and checks the signature. Finding nothing amiss, he passes the signed message to Bob.
(5) Bob checks the signature on the innocuous message, confirming that the message came from Alice.

This protocol makes it very difficult for Alice and Trent to collude and produce a document stamped with a different time than the actual one. Trent cannot forward-date a document for Alice, since that would require knowing in advance what document request came before it. Even if he could fake that, he would have to know what document request came before that, and so on. He cannot back-date a document, because the timestamp must be embedded in the timestamps of the document issued immediately after, and that document has already been issued. The only possible way to break this scheme is to invent a fictitious chain of documents both before and after Alice's document, long enough to exhaust the patience of anyone challenging the timestamp.

Distributed Protocol

People die; timestamps get lost. Many things could happen between the timestamping and the challenge to make it impossible for Alice to get a copy of I_{n-1}'s timestamp. This problem could be alleviated by embedding the previous 10 people's timestamps into Alice's, and then sending Alice the identities of the next 10 people. Alice has a greater chance of finding people who still have their timestamps.

Along a similar line, the following protocol does away with Trent altogether.

(1) Using H_n as input, Alice generates a string of random values using a cryptographically secure pseudo-random-number generator:
$V_1, V_2, V_3, ...V_k$
(2) Alice interprets each of these values as the identification, *I*, of another person. She sends H_n to each of these people.
(3) Each of these people attaches the date and time to the hash, signs the result, and sends it back to Alice.
(4) Alice collects and stores all the signatures as the timestamp.

The cryptographically secure pseudo-random-number generator in step (1) prevents Alice from deliberately choosing corrupt *I*s as verifiers. Even if she makes trivial changes in her document in an attempt to construct a set of corrupt *I*s, her chances of getting away with this are negligible. The hash function randomizes the *I*s; Alice cannot force them.

This protocol works because the only way for Alice to fake a timestamp would be to convince all of the *k* people to cooperate. Since she chose them at random in step (1), the odds against this are very high. The more corrupt society is, the higher a number *k* should be.

Additionally, there should be some mechanism for dealing with people who can't promptly return the timestamp. Some subset of *k* is all that would be required for a valid timestamp. The details depend on the implementation.

Further Work

Further improvements to timestamping protocols are presented in [92]. The authors use binary trees to increase the number of timestamps that depend on a given timestamp, reducing even further the possibility that someone could create a chain of fictitious timestamps. They also recommend publishing a hash of the day's timestamps in a public place, such as a newspaper. This serves a function similar to sending the hash to random people in the distributed protocol. In fact, a timestamp has appeared in every Sunday's *New York Times* since 1992.

These timestamping protocols are patented [684, 685, 686]. A Bellcore spin-off company called Surety Technologies owns the patents and markets a Digital Notary System to support these

colluding and timestamping a document with any time that they want.

Improved Arbitrated Solution

One-way hash functions and digital signatures can clear up most of these problems easily:

(1) Alice produces a one-way hash of the document.
(2) Alice transmits the hash to Trent.
(3) Trent appends the date and time he received the hash onto the hash and then digitally signs the result.
(4) Trent sends the signed hash with timestamp back to Alice.

This solves every problem but the last. Alice no longer has to worry about revealing the contents of her document; the hash is sufficient. Trent no longer has to store copies of the document (or even of the hash), so the massive storage requirements and security problems are solved (remember, one-way hash functions don't have a key). Alice can immediately examine the signed timestamped hash she receives in step (4), so she will immediately catch any transmission errors. The only problem remaining is that Alice and Trent can still collude to produce any timestamp they want.

Linking Protocol

One way to solve this problem is to link Alice's timestamp with timestamps previously generated by Trent. These timestamps will most probably be generated for people other than Alice. Since the order that Trent receives the different timestamp requests can't be known in advance, Alice's timestamp must have occurred after the previous one. And since the request that came after is linked with Alice's timestamp, then hers must have occurred before. This sandwiches Alice's request in time.

If A is Alice's name, the hash value that Alice wants timestamped is H_n, and the previous time stamp is T_{n-1}, then the protocol is:

(1) Alice sends Trent H_n and A.
(2) Trent sends back to Alice:

$$T_n = S_K(n,A,H_n,t_n,I_{n-1},H_{n-1},T_{n-1},L_n)$$

where L_n consists of the following hashed linking information:

$$L_n = H(I_{n-1},H_{n-1},T_{n-1},L_{n-1})$$

S_K indicates that the message is signed with Trent's private key. Alice's name identifies her as the originator of the request. The parameter n indicates the sequence of the request: This is the nth timestamp Trent has issued. The parameter t_n is the time. The additional information is the identification, original hash, time, and hashed timestamp of the previous document Trent stamped.
(3) After Trent stamps the next document, he sends Alice the identification of the originator of that document: I_{n+1}.

If someone challenges Alice's timestamp, she just contacts the originators of the previous and following documents: I_{n-1} and I_{n+1}. If their documents are called into question, they can get in touch with I_{n-2} and I_{n+2}, and so on. Every person can show that their document was timestamped after the one that came before and before the one that came after.

Chapter 4
Intermediate Protocols

4.1 Timestamping Services

In many situations, people need to certify that a document existed on a certain date. Think about a copyright or patent dispute: The party that produces the earliest copy of the disputed work wins the case. With paper documents, notaries can sign and lawyers can safeguard copies. If a dispute arises, the notary or the lawyer testifies that the letter existed on a certain date.

In the digital world, it's far more complicated. There is no way to examine a digital document for signs of tampering. It can be copied and modified endlessly without anyone being the wiser. It's trivial to change the date stamp on a computer file. No one can look at a digital document and say: "Yes, this document was created before November 4, 1952."

Stuart Haber and W. Scott Stornetta at Bellcore thought about the problem [682, 683, 92]. They wanted a digital timestamping protocol with the following properties:

— The data itself must be timestamped, without any regard to the physical medium on which it resides.
— It must be impossible to change a single bit of the document without that change being apparent.
— It must be impossible to timestamp a document with a date and time different from the present one.

Arbitrated Solution

This protocol uses Trent, who has a trusted timestamping service, and Alice, who wishes to timestamp a document.

(1) Alice transmits a copy of the document to Trent.
(2) Trent records the date and time he received the document and retains a copy of the document for safekeeping.

Now, if anyone calls into question Alice's claim of when the document was created, she just has to call up Trent. He will produce his copy of the document and verify that he received the document on the date and time stamped.

This protocol works, but has some obvious problems. First, there is no privacy. Alice has to give a copy of the document to Trent. Anyone listening in on the communications channel could read it. She could encrypt it, but still the document has to sit in Trent's database. Who knows how secure that database is?

Second, the database itself would have to be huge. And the bandwidth requirements to send large documents to Trent would be unwieldy.

The third problem has to do with the potential errors. An error in transmission, or an electromagnetic bomb detonating somewhere in Trent's central computers, could completely invalidate Alice's claim of a timestamp.

And fourth, there might not be someone as honest as Trent to run the timestamping service. Maybe Alice is using Bob's Timestamp and Taco Stand. There is nothing to stop Alice and Bob from

Eve being able to read these mail messages, then they should encrypt their transmissions to each other.)

(12) Alice and Bob repeat step (11) for the second bits of all 2*n* DES keys, the third bits, and so on, until all the bits of all the DES keys have been transferred.

(13) Alice and Bob decrypt the remaining halves of the message pairs. Alice has a valid receipt from Bob, and Bob can XOR any key pair to get the original message encryption key.

(14) Alice and Bob exchange the private keys used during the oblivious transfer protocol and each verifies that the other did not cheat.

Steps (5) through (8) for Bob, and steps (9) through (12) for both Alice and Bob, are the same as the contract-signing protocol. The twist is all of Alice's dummy messages. They give Bob some way of checking the validity of her oblivious transfer in step (10), which forces her to stay honest during steps (11) through (13). And, as with the simultaneous contract-signing protocol, both a left and a right half of one of Alice's message pairs are required to complete the protocol.

5.9 Simultaneous Exchange of Secrets

Alice knows secret *A;* Bob knows secret *B*. Alice is willing to tell Bob *A*, if Bob tells her *B*. Bob is willing to tell Alice *B*, if Alice tells him *A*. This protocol, observed in a schoolyard, does not work:

(1) Alice: "I'll tell if you tell me first."
(2) Bob: "I'll tell if you tell me first."
(3) Alice: "No, you first."
(4) Bob: "Oh, all right." Bob whispers.
(5) Alice: "Ha! I won't tell you."
(6) Bob: "That's not fair."

Cryptography can make it fair. The previous two protocols are implementations of this more general protocol, one that lets Alice and Bob exchange secrets simultaneously [529]. Rather than repeat the whole protocol, I'll sketch the modifications to the certified mail protocol.

Alice performs steps (1) through (4) using *A* as the message. Bob goes through similar steps using *B* as his message. Alice and Bob perform the oblivious transfer in step (9), decrypt the halves they can in step (10), and go through the iterations in steps (11) and (12). If they are concerned about Eve, they should encrypt their messages. Finally, both Alice and Bob decrypt the remaining halves of the message pairs and XOR any key pair to get the original message encryption key.

This protocol allows Alice and Bob to exchange secrets simultaneously, but says nothing about the quality of the secrets exchanged. Alice could promise Bob the solution to the Minotaur's labyrinth, but actually send him a map of Boston's subway system. Bob will get whatever secret Alice sends him. Other protocols are [1286,195,991,1524,705,753,259,358,415].

protocol at a point when Bob would have to spend a year's worth of computing power before she is really committed, that's a problem. The real difficulty here is the lack of a near-term deadline by which the process cleanly terminates with either both or neither party bound.

These problems also apply to the protocols in Sections 5.8 and 5.9.

5.8 Digital Certified Mail

The same simultaneous oblivious transfer protocol used for contract signing works, with some modifications, for computer certified mail [529]. Suppose Alice wants to send a message to Bob, but she does not want him to read it without signing a receipt. Surly postal workers handle this process in real life, but the same thing can be done with cryptography. Whitfield Diffie first discussed this problem in [490].

At first glance, the simultaneous contract-signing protocol can do this. Alice simply encrypts her message with a DES key. Her half of the protocol can be something like: "This is the left half of the DES key: 32f5," and Bob's half can be something like: "This is the left half of my receipt." Everything else stays the same.

To see why this won't work, remember that the protocol hinges on the fact that the oblivious transfer in step (5) keeps both parties honest. Both of them know that they sent the other party a valid half, but neither knows which. They don't cheat in step (8) because the odds of getting away with it are miniscule. If Alice is sending Bob not a message but half of a DES key, Bob can't check the validity of the DES key in step (6). Alice can still check the validity of Bob's receipt, so Bob is still forced to be honest. Alice can freely send Bob some garbage DES key, and he won't know the difference until she has a valid receipt. Tough luck, Bob.

Getting around this problem requires some adjustment of the protocol:

(1) Alice encrypts her message using a random DES key, and sends the message to Bob.
(2) Alice generates n pairs of DES keys. The first key of each pair is generated at random; the second key of each pair is the XOR of the first key and the message encryption key.
(3) Alice encrypts a dummy message with each of her $2n$ keys.
(4) Alice sends the whole pile of encrypted messages to Bob, making sure he knows which messages are which halves of which pairs.
(5) Bob generates n pairs of random DES keys.
(6) Bob generates a pair of messages that indicates a valid receipt. "This is the left half of my receipt" and "this is the right half of my receipt" are good candidates, with the addition of some kind of random-bit string. He makes n receipt pairs, each numbered. As with the previous protocol, the receipt is considered valid if Alice can produce both halves of a receipt (with the same number) and all of her encryption keys.
(7) Bob encrypts each of his message pairs with DES key pairs, the ith message pair with the ith key pair, the left message with the left key in the pair, and the right message with the right key in the pair.
(8) Bob sends his pile of message pairs to Alice, making sure that Alice knows which messages are which halves of which pairs.
(9) Alice and Bob send each other every key pair using the oblivious transfer protocol. That is, Alice sends Bob either the key used to encrypt the left message or the key used to encrypt the right message, for each of the n pairs. Bob does the same. They can either alternate sending halves or one can send n and then the other—it doesn't matter. Now both Alice and Bob have one key in each key pair, but neither knows which halves the other has.
(10) Both Alice and Bob decrypt the halves they can and make sure that the decrypted messages are valid.
(11) Alice and Bob send each other the first bits of all $2n$ DES keys. (If they are worried about

matter. Now both Alice and Bob have one key in each key pair, but neither knows which halves the other one has.

(6) Both Alice and Bob decrypt the message halves that they can, using the keys they received. They make sure that the decrypted messages are valid.

(7) Alice and Bob send each other the first bits of all 2*n* DES keys.

(8) Alice and Bob repeat step (7) for the second bits of all 2*n* DES keys, the third bits, and so on, until all the bits of all the DES keys have been transferred.

(9) Alice and Bob decrypt the remaining halves of the message pairs and the contract is signed.

(10) Alice and Bob exchange the private keys used during the oblivious transfer protocol in step (5) and each verifies that the other did not cheat.

Why do Alice and Bob have to go through all this work? Let's assume Alice wants to cheat and see what happens. In steps (4) and (5), Alice could disrupt the protocol by sending Bob nonsense bit strings. Bob would catch this in step (6), when he tried to decrypt whatever half he received. Bob could then stop safely, before Alice could decrypt any of Bob's message pairs.

If Alice were very clever, she could only disrupt half the protocol. She could send one half of each pair correctly, but send a gibberish string for the other half. Bob has only a 50 percent chance of receiving the correct half, so half the time Alice could cheat. However, this only works if there is one key pair. If there were only two pairs, this sort of deception would succeed 25 percent of the time. That is why *n* should be large. Alice has to guess correctly the outcome of *n* oblivious transfer protocols; she has a 1 in 2^n chance of doing this. If $n = 10$, Alice has a 1 in 1024 chance of deceiving Bob.

Alice could also send Bob random bits in step (8). Perhaps Bob won't know that she is sending him random bits until he receives the whole key and tries to decrypt the message halves. But again, Bob has probability on his side. He has already received half of the keys, and Alice does not know which half. If *n* is large enough, Alice is sure to send him a nonsense bit to a key he has already received and he will know immediately that she is trying to deceive him.

Maybe Alice will just go along with step (8) until she has enough bits of the keys to mount a brute-force attack and then stop transmitting bits. DES has a 56-bit-long key. If she receives 40 of the 56 bits, she only has to try 2^{16}, or 65,536, keys in order to read the message—a task certainly within the realm of a computer's capabilities. But Bob will have exactly the same number of bits of her keys (or, at worst, one bit less), so he can do the same thing. Alice has no real choice but to continue the protocol.

The basic point is that Alice has to play fairly, because the odds of fooling Bob are just too small. At the end of the protocol, both parties have *n* signed message pairs, any one of which is sufficient for a valid signature.

There is one way Alice can cheat; she can send Bob identical messages in Step (5). Bob can't detect this until after the protocol is finished, but he can use a transcript of the protocol to convince a judge of Alice's duplicity.

There are two weaknesses with protocols of this type [138]. First, it's a problem if one of the parties has significantly more computing power than the other. If, for example, Alice can mount a brute-force attack faster than Bob can, then she can stop sending bits early in step (8), and figure out Bob's keys herself. Bob, who cannot do the same in a reasonable amount of time, will not be happy.

Second, it's a problem if one of the parties stops the protocol early. If Alice abruptly stops the protocol, both face similar computational efforts, but Bob does not have any real legal recourse. If, for example, the contract specifies that she do something in a week, and Alice terminates the

(1) Alice and Bob agree on a date by which the signing protocol should be completed.
(2) Alice and Bob decide on a probability difference that they are willing to live with. For example, Alice might decide that she is not willing to be bound with a greater probability than 2 percent over Bob's probability. Call Alice's difference *a;* call Bob's difference *b*.
(3) Alice sends Bob a signed message with $p = a$.
(4) Bob sends Alice a signed message with $p = a + b$.
(5) Let p be the probability of the message Alice received in the previous step from Bob. Alice sends Bob a signed message with $p' = p + a$ or 1, whichever is smaller.
(6) Let p be the probability of the message Bob received in the previous step from Alice. Bob sends Alice a signed message with $p' = p + b$ or 1, whichever is smaller.
(7) Alice and Bob continue alternating steps (5) and (6) until both have received messages with $p = 1$ or until the date agreed to in step (1) has passed.

As the protocol proceeds, both Alice and Bob agree to be bound to the contract with a greater and greater probability. For example, Alice might define her *a* as 2 percent and Bob might define his *b* as 1 percent. (It would be nice if they had chosen larger increments; we will be here for a while.) Alice's first message might state that she is bound with 2 percent probability. Bob might respond that he is bound with 3 percent probability. Alice's next message might state that she is bound with 5 percent probability and so on, until both are bound with 100 percent probability.

If both Alice and Bob complete the protocol by the completion date, all is well. Otherwise, either party can take the contract to the judge, along with the other party's last signed message. The judge then randomly chooses a value between 0 and 1 before seeing the contract. If the value is less than the probability the other party signed, then both parties are bound. If the value is greater than the probability, then both parties are not bound. (The judge then saves the value, in case he has to rule on another matter regarding the same contract.) This is what is meant by being bound to the contract with probability *p*.

That's the basic protocol, but it can have more complications. The judge can rule in the absence of one of the parties. The judge's ruling either binds both or neither party; in no situation is one party bound and the other one not. Furthermore, as long as one party is willing to have a slightly higher probability of being bound than the other (no matter how small), the protocol will terminate.

Simultaneous Contract Signing without an Arbitrator (Using Cryptography)

This cryptographic protocol uses the same baby-step approach [529]. DES is used in the description, although any symmetric algorithm will do.

(1) Both Alice and Bob randomly select $2n$ DES keys, grouped in pairs. The pairs are nothing special; they are just grouped that way for the protocol.
(2) Both Alice and Bob generate n pairs of messages, L_i and R_i: "This is the left half of my ith signature" and "This is the right half of my ith signature," for example. The identifier, i, runs from 1 to n. Each message will probably also include a digital signature of the contract and a timestamp. The contract is considered signed if the other party can produce both halves, L_i and R_i, of a single signature pair.
(3) Both Alice and Bob encrypt their message pairs in each of the DES key pairs, the left message with the left key in the pair and the right message with the right key in the pair.
(4) Alice and Bob send each other their pile of $2n$ encrypted messages, making clear which messages are which halves of which pairs.
(5) Alice and Bob send each other every key pair using the oblivious transfer protocol for each pair. That is, Alice sends Bob either the key used to encrypt the left message or the key used to encrypt the right message, independently, for each of the n pairs. Bob does the same. They can either alternate sending halves or one can send 100 and then the other—it doesn't

(1) Alice signs a copy of the contract and sends it to Trent.
(2) Bob signs a copy of the contract and sends it to Trent.
(3) Trent sends a message to both Alice and Bob indicating that the other has signed the contract.
(4) Alice signs two copies of the contract and sends them to Bob.
(5) Bob signs both copies of the contract, keeps one for himself, and sends the other to Alice.
(6) Alice and Bob both inform Trent that they each have a copy of the contract signed by both of them.
(7) Trent tears up his two copies of the contract with only one signature each.

This protocol works because Trent prevents either of the parties from cheating. If Bob were to refuse to sign the contract in step (5), Alice could appeal to Trent for a copy of the contract already signed by Bob. If Alice were to refuse to sign in step (4), Bob could do the same. When Trent indicates that he received both contracts in step (3), both Alice and Bob know that the other is bound by the contract. If Trent does not receive both contracts in steps (1) and (2), he tears up the one he received and neither party is bound.

Simultaneous Contract Signing without an Arbitrator (Face-to-Face)

If Alice and Bob were sitting face-to-face, they could sign the contract this way [1244]:

(1) Alice signs the first letter of her name and passes the contract to Bob.
(2) Bob signs the first letter of his name and passes the contract to Alice.
(3) Alice signs the second letter of her name and passes the contract to Bob.
(4) Bob signs the second letter of his name and passes the contract to Alice.
(5) This continues until both Alice and Bob have signed their entire names.

If you ignore the obvious problem with this protocol (Alice has a longer name than Bob), it works just fine. After signing only one letter, Alice knows that no judge will bind her to the terms of the contract. But the letter is an act of good faith, and Bob responds with a similar act of good faith.

After each party has signed several letters, a judge could probably be convinced that both parties had signed the contract. The details are murky, though. Surely they are not bound after only the first letter; just as surely they are bound after they sign their entire names. At what point in the protocol do they become bound? After signing one-half of their names? Two-thirds of their names? Three-quarters?

Since neither Alice nor Bob is certain of the exact point at which she or he is bound, each has at least some fear that she or he is bound throughout the protocol. At no point can Bob say: “You signed four letters and I only signed three. You are bound but I am not.” Bob has no reason not to continue with the protocol. Furthermore, the longer they continue, the greater the probability that a judge will rule that they are bound. Again, there is no reason not to continue with the protocol. After all, they both wanted to sign the contract; they just didn’t want to sign before the other one.

Simultaneous Contract Signing without an Arbitrator (Not Face-to-Face)

This protocol uses the same sort of uncertainty [138]. Alice and Bob alternate taking baby steps toward signing until both have signed.

In the protocol, Alice and Bob exchange a series of signed messages of the form: “I agree that with probability *p*, I am bound by this contract.”

The recipient of this message can take it to a judge and, with probability *p*, the judge will consider the contract to be signed.

(5) Bob gets one of Alice's messages encrypted with the proper DES key and the other one encrypted with the gibberish DES key. When Bob decrypts each of them with his DES key, he can read one of them; the other just looks like gibberish to him.

Bob now has one of the two messages from Alice and Alice does not know which one he was able to read successfully. Unfortunately, if the protocol stopped here it would be possible for Alice to cheat. Another step is necessary.

(6) After the protocol is complete and both possible results of the transfer are known, Alice must give Bob her private keys so that he can verify that she did not cheat. After all, she could have encrypted the same message with both keys in step (4).

At this point, of course, Bob can figure out the second message.

The protocol is secure against an attack by Alice because she has no way of knowing which of the two DES keys is the real one. She encrypts them both, but Bob only successfully recovers one of them—until step (6). It is secure against an attack by Bob because, before step (6), he cannot get Alice's private keys to determine the DES key that the other message was encrypted in. This may still seem like nothing more than a more complicated way to flip coins over a modem, but it has extensive implications when used in more complicated protocols.

Of course, nothing stops Alice from sending Bob two completely useless messages: "Nyah Nyah" and "You sucker." This protocol guarantees that Alice sends Bob one of two messages; it does nothing to ensure that Bob wants to receive either of them.

Other oblivious transfer protocols are found in the literature. Some of them are noninteractive, meaning that Alice can publish her two messages and Bob can learn only one of them. He can do this on his own; he doesn't have to communicate with Alice [105].

No one really cares about being able to do oblivious transfer in practice, but the notion is an important building block for other protocols. Although there are many types of oblivious transfer—I have two secrets and you get one; I have *n* secrets and you get one; I have one secret which you get with probability 1/2; and so on—they are all equivalent [245,391,395].

5.6 Oblivious Signatures

Honestly, I can't think of a good use for these, but there are two kinds [346]:

1. Alice has *n* different messages. Bob can choose one of the *n* messages for Alice to sign, and Alice will have no way of knowing which one she signed.
2. Alice has one message. Bob can choose one of *n* keys for Alice to use in signing the message, and Alice will have no way of knowing which key she used.

It's a neat idea; I'm sure it has a use somewhere.

5.7 Simultaneous Contract Signing

Contract Signing with an Arbitrator

Alice and Bob want to enter into a contract. They've agreed on the wording, but neither wishes to sign unless the other signs as well. Face to face, this is easy: Both sign together. Over a distance, they could use an arbitrator.

4,759,063	7/19/88	Blind Signature Systems [323]
4,759,064	7/19/88	Blind Unanticipated Signature Systems [324]
4,914,698	3/3/90	One-Show Blind Signature Systems [326]
4,949,380	8/14/90	Returned-Value Blind Signature Systems [328]
4,991,210	2/5/91	Unpredictable Blind Signature Systems [331]

5.5 Oblivious Transfer

Cryptographer Bob is desperately trying to factor a 500-bit number, *n*. He knows it's the product of five 100-bit numbers, but nothing more. (This is a problem. If he can't recover the key he'll have to work overtime and he'll miss his weekly mental poker game with Alice.)

What do you know? Here comes Alice now:

> "I happen to know one factor of the number," she says, "and I'll sell it to you for $100. That's a dollar a bit." To show she's serious, she uses a bit-commitment scheme and commits to each bit individually.
>
> Bob is interested, but has only $50. Alice is unwilling to lower her price and offers to sell Bob half the bits for half the price. "It'll save you a considerable amount of work," she says.
>
> "But how do I know that your number is actually a factor of *n?* If you show me the number and let me verify that it is a factor, then I will agree to your terms," says Bob.
>
> They are at an impasse. Alice cannot convince Bob that her number is a factor of *n* without revealing it, and Bob is unwilling to buy 50 bits of a number that could very well be worthless.

This story, stolen from Joe Kilian [831], introduces the concept of **oblivious transfer**. Alice transmits a group of messages to Bob. Bob receives some subset of those messages, but Alice has no idea which ones he receives. This doesn't completely solve the problem, however. After Bob has received a random half of the bits, Alice has to convince him that the bits she sent are part of a factor of *n*, using a zero-knowledge proof.

In the following protocol, Alice will send Bob one of two messages. Bob will receive one, and Alice will not know which.

> **(1)** Alice generates two public-key/private-key key pairs, or four keys in all. She sends both public keys to Bob.
> **(2)** Bob chooses a key in a symmetric algorithm (DES, for example). He chooses one of Alice's public keys and encrypts his DES key with it. He sends the encrypted key to Alice without telling her which of her public keys he used to encrypt it.
> **(3)** Alice decrypts Bob's key twice, once with each of her private keys. In one of the cases, she uses the correct key and successfully decrypts Bob's DES key. In the other case, she uses the wrong key and only manages to generate a meaningless pile of bits that nonetheless look like a random DES key. Since she does not know the correct plaintext, she has no idea which is which.
> **(4)** Alice encrypts both of her messages, each with a different one of the DES keys she generated in the previous step (one real and one meaningless) and sends both of them to Bob.

mega-document. In step (8), BOB strips off all the blinding factors. ALICE's signature is acceptable only if it is a valid signature of the product of *n*/2 identical documents. To cheat BOB has to be able to guess exactly which subset ALICE will challenge; the odds are much smaller than the odds of guessing which one document ALICE won't challenge.

BOB has another way to cheat. He can generate two different documents, one that ALICE is willing to sign and one that ALICE is not. Then he can find two different blinding factors that transform each document into the same blinded document. That way, if ALICE asks to examine the document, BOB gives her the blinding factor that transforms it into the benign document. If ALICE doesn't ask to see the document and signs it, he uses the blinding factor that transforms it into the malevolent document. While this is theoretically possible, the mathematics of the particular algorithms involved make the odds of BOB's being able to find such a pair negligibly small. In fact, it can be made as small as the odds of Bob being able to produce the signature on an arbitrary message himself. This issue is discussed further in Section 23.12.

Patents

Chaum has patents for several flavors of blind signatures (see Table 5.1).

5.4 Identity-Based Public-Key Cryptography

Alice wants to send a secure message to Bob. She doesn't want to get his public key from a key server; she doesn't want to verify some trusted third party's signature on his public-key certificate; and she doesn't even want to store Bob's public key on her own computer. She just wants to send him a secure message.

Identity-based cryptosystems, sometimes called Non-Interactive Key Sharing (NIKS) systems, solve this problem [1422]. Bob's public key is based on his name and network address (or telephone number, or physical street address, or whatever). With normal public-key cryptography, Alice needs a signed certificate that associates Bob's public key with his identity. With identity-based cryptography, Bob's public key *is* his identity. This is a really cool idea, and about as ideal as you can get for a mail system: If Alice knows Bob's address, she can send him secure mail. It makes the cryptography about as transparent as possible.

The system is based on Trent issuing private keys to users based on their identity. If Alice's private key is compromised, she has to change some aspect of her identity to get another one. A serious problem is designing a system in such a way that a collusion of dishonest users cannot forge a key.

A lot of work has been done on the mathematics of these sorts of schemes—most of it in Japan—which turn out to be infuriatingly complicated to make secure. Many of the proposed solutions involve Trent choosing a random number for each user—in my opinion this defeats the real point of the system. Some of the algorithms discussed in Chapters 19 and 20 can be identity-based. For details, algorithms, and cryptanalysis, see [191,1422,891,1022,1515,1202,1196,908,692,674,1131,1023,1516,1536,1544,63, 1210,314,313,1545,1539,1543,933,1517,748,1228]. An algorithm that does not rely on any random numbers is [1035]. The system discussed in [1546,1547,1507] is insecure against a chosen-public-key attack; so is the system proposed as NIKS-TAS [1542,1540,1541,993,375,1538]. Honestly, nothing proposed so far is both practical and secure.

TABLE 5.1
Chaum's Blind Signature Patents

U.S. PATENT #	DATE	TITLE

court system is effective, the penalty for getting caught once will more than wipe out the gains from the other nine times.

If the Department of Immigration wants to increase the odds of catching smugglers, they have to search more people. If they want to decrease the odds, they have to search fewer people. By manipulating the probabilities, they control how successful the protocol is in catching smugglers.

The blind signature protocol works in a similar manner. Bob will be given a large pile of different blinded documents. He will **open**, that is examine, all but one and then sign the last.

Think of the blinded document as being in an envelope. The process of blinding the document is putting the document in an envelope and the process of removing the blinding factor is opening the envelope. When the document is in an envelope, nobody can read it. The document is signed by having a piece of carbon paper in the envelope: When the signer signs the envelope, his signature goes through the carbon paper and signs the document as well.

This scenario involves a group of counterintelligence agents. Their identities are secret; not even the counterintelligence agency knows who they are. The agency's director wants to give each agent a signed document stating: "The bearer of this signed document, (insert agent's cover name here), has full diplomatic immunity." Each of the agents has his own list of cover names, so the agency can't just hand out signed documents. The agents do not want to send their cover names to the agency; the enemy might have corrupted the agency's computer. On the other hand, the agency doesn't want to blindly sign any document an agent gives it. A clever agent might substitute a message like: "Agent (name) has retired and collects a million-dollar-a-year pension. Signed, Mr. President." In this case, blind signatures could be useful.

Assume that all the agents have 10 possible cover names, which they have chosen themselves and which no one else knows. Also assume that the agents don't care under which cover name they are going to get diplomatic immunity. Also assume that the agency's computer is the Agency's Large Intelligent Computing Engine, or ALICE, and that our particular agent is the Bogota Operations Branch: BOB.

(1) BOB prepares n documents, each using a different cover name, giving himself diplomatic immunity.
(2) BOB blinds each of these documents with a different blinding factor.
(3) BOB sends the n blinded documents to ALICE.
(4) ALICE chooses $n - 1$ documents at random and asks BOB for the blinding factors for each of those documents.
(5) BOB sends ALICE the appropriate blinding factors.
(6) ALICE opens (i.e., she removes the blinding factor) $n - 1$ documents and makes sure they are correct—and not pension authorizations.
(7) ALICE signs the remaining document and sends it to BOB.
(8) Agent removes the blinding factor and reads his new cover name: "The Crimson Streak." The signed document gives him diplomatic immunity under that name.

This protocol is secure against BOB cheating. For him to cheat, he would have to predict accurately which document ALICE would not examine. The odds of him doing this are 1 in n—not very good. ALICE knows this and feels confident signing a document that she is not able to examine. With this one document, the protocol is the same as the previous completely blinded signature protocol and maintains all of its properties of anonymity.

There is a trick that makes BOB's chance of cheating even smaller. In step (4), ALICE randomly chooses $n/2$ of the documents to challenge, and BOB sends her the appropriate blinding factors in step (5). In step (7), ALICE multiplies together all of the unchallenged documents and signs the

signer can *almost*, but not exactly, know what he is signing. But first things first.

Completely Blind Signatures

Bob is a notary public. Alice wants him to sign a document, but does not want him to have any idea what he is signing. Bob doesn't care what the document says; he is just certifying that he notarized it at a certain time. He is willing to go along with this.

(1) Alice takes the document and multiplies it by a random value. This random value is called a **blinding factor**.
(2) Alice sends the blinded document to Bob.
(3) Bob signs the blinded document.
(4) Alice divides out the blinding factor, leaving the original document signed by Bob.

This protocol only works if the signature function and multiplication are commutative. If they are not, there are other ways to modify the document other than by multiplying. Some relevant algorithms appear in Section 23.12. For now, assume that the operation is multiplication and all the math works.

Can Bob cheat? Can he collect any information about the document that he is signing? If the blinding factor is truly random and makes the blinded document truly random, he cannot. The blinded document Bob signs in step (2) looks nothing like the document Alice began with. The blinded document with Bob's signature on it in step (3) looks nothing like the signed document at the end of step (4). Even if Bob got his hands on the document, with his signature, after completing the protocol, he cannot prove (to himself or to anyone else) that he signed it in that particular protocol. He knows that his signature is valid. He can, like anyone else, verify his signature. However, there is no way for him to correlate any information he received during the signing protocol with the signed document. If he signed a million documents using this protocol, he would have no way of knowing in which instance he signed which document.

The properties of completely blind signatures are:

1. Bob's signature on the document is valid. The signature is a proof that Bob signed the document. It will convince Bob that he signed the document if it is ever shown to him. It also has all of the other properties of digital signatures discussed in Section 2.6.
2. Bob cannot correlate the signed document with the act of signing the document. Even if he keeps records of every blind signature he makes, he cannot determine when he signed any given document.

Eve, who is in the middle, watching this protocol, has even less information than Bob.

Blind Signatures

With the completely blind signature protocol, Alice can have Bob sign anything: "Bob owes Alice a million dollars," "Bob owes Alice his first-born child," "Bob owes Alice a bag of chocolates." The possibilities are endless. This protocol isn't useful in many applications.

However, there is a way that Bob can know what he is signing, while still maintaining the useful properties of a blind signature. The heart of this protocol is the cut-and-choose technique. Consider this example. Many people enter this country every day, and the Department of Immigration wants to make sure they are not smuggling cocaine. The officials could search everyone, but instead they use a probabilistic solution. They will search one-tenth of the people coming in. One person in ten has his belongings inspected; the other nine get through untouched. Chronic smugglers will get away with their misdeeds most of the time, but they have a 10 percent chance of getting caught. And if the

The Multiple Identity Fraud

There are other possible abuses to zero-knowledge proofs of identity, also discussed in [485,120]. In some implementations, there is no check when an individual registers a public key. Hence, Alice can have several private keys and, therefore, several identities. This can be a great help if she wants to commit tax fraud. Alice can also commit a crime and disappear. First, she creates and publishes several identities. One of them she doesn't use. Then, she uses that identity once and commits a crime so that the person who identifies her is the witness. Then, she immediately stops using that identity. The witness knows the identity of the person who committed the crime, but if Alice never uses that identity again—she's untraceable.

To prevent this, there has to be some mechanism by which each person has only one identity. In [120] the authors suggest the bizarre idea of tamperproof babies who are impossible to clone and contain a unique number as part of their genetic code. They also suggested having each baby apply for an identity at birth. (Actually, the parents would have to do this as the baby would be otherwise occupied.) This could easily be abused; parents could apply for multiple identities at the child's birth. In the end, the uniqueness of an individual is based on trust.

Renting Passports

Alice wants to travel to Zaire, but that government won't give her a visa. Carol offers to rent her identity to Alice. (Bob offered first, but there were some obvious problems.) Carol sells Alice her private key and Alice goes off to Zaire pretending to be Carol.

Carol has not only been paid for her identity, but now she has a perfect alibi. She commits a crime while Alice is in Zaire. "Carol" has proved her identity in Zaire; how could she commit a crime back home?

Of course, Alice is free to commit crimes as well. She does so either before she leaves or after she returns, near Carol's home. First she identifies herself as Carol (she has Carol's private key, so she can easily do that), then she commits a crime and runs away. The police will come looking for Carol. Carol will claim she rented her identity to Alice, but who would believe such a nonsensical story?

The problem is that Alice isn't really proving her identity; she is proving that she knows a piece of secret information. It is the link between that information and the person it belongs to that is being abused. The tamperproof baby solution would protect against this type of fraud, as would a police state where all citizens would have to prove their identity very frequently (at the end of each day, at each street corner, etc.). Biometric methods—fingerprints, retinal scanning, voiceprints, and so on—may help solve this problem.

Proofs of Membership

Alice wants to prove to Bob that she is a member of some super-secret organization, but she does not want to reveal her identity. This problem is similar but different to proving identity, and has also been studied [887,906,907,1201,1445]. Some solutions are related to the problem of group signatures (see Section 4.6).

5.3 Blind Signatures

An essential feature of digital signature protocols is that the signer knows what he is signing. This is a good idea, except when we want the reverse.

We might want people to sign documents without ever seeing their contents. There are ways that a

In reality, Kasparov is playing Karpov and Alice is simply acting as the middleman, mimicking the moves of each grandmaster on the other's board. However, if neither Karpov nor Kasparov knows about the other's presence, each will be impressed with Alice's play.

This kind of fraud can be used against zero-knowledge proofs of identity [485,120]. While Alice is proving her identity to Mallory, Mallory can simultaneously prove to Bob that he is Alice.

The Mafia Fraud

When discussing his zero-knowledge identification protocol, Adi Shamir [1424] said: "I could go to a Mafia-owned store a million successive times and they will still not be able to misrepresent themselves as me."

Here's how the Mafia can. Alice is eating at Bob's Diner, a Mafia-owned restaurant. Carol is shopping at Dave's Emporium, an expensive jewelry store. Bob and Carol are both members of the Mafia and are communicating by a secret radio link. Alice and Dave are unaware of the fraud.

At the end of Alice's meal, when she is ready to pay and prove her identity to Bob, Bob signals Carol that the fraud is ready to begin. Carol chooses some expensive diamonds and gets ready to prove her identity to Dave. Now, as Alice proves her identity to Bob, Bob radios Carol and Carol performs the same protocol with Dave. When Dave asks a question in the protocol, Carol radios the question back to Bob, and Bob asks it of Alice. When Alice answers, Bob radios the correct answer to Carol. Actually, Alice is just proving her identity to Dave, and Bob and Carol are simply sitting in the middle of the protocol passing messages back and forth. When the protocol finishes, Alice has proved herself to Dave and has purchased some expensive diamonds (which Carol disappears with).

The Terrorist Fraud

If Alice is willing to collaborate with Carol, they can also defraud Dave. In this protocol, Carol is a well-known terrorist. Alice is helping her enter the country. Dave is the immigration officer. Alice and Carol are connected by a secret radio link.

When Dave asks Carol questions as part of the zero-knowledge protocol, Carol radios them back to Alice, who answers them herself. Carol recites these answers to Dave. In reality, Alice is proving her identity to Dave, with Carol acting as a communications path. When the protocol finishes, Dave thinks that Carol is Alice and lets her into the country. Three days later, Carol shows up at some government building with a minivan full of explosives.

Suggested Solutions

Both the Mafia and Terrorist frauds are possible because the conspirators can communicate via a secret radio. One way to prevent this requires all identifications to take place inside Faraday cages, which block all electromagnetic radiation. In the terrorist example, this assures immigration officer Dave that Carol was not receiving her answers from Alice. In the Mafia example, Bob could simply build a faulty Faraday cage in his restaurant, but jeweler Dave would have a working one; Bob and Carol would not be able to communicate. To solve the Chess Grandmaster Problem, Alice should be forced to sit in her seat until the end of a game.

Thomas Beth and Yvo Desmedt proposed another solution, one using accurate clocks [148]. If each step in the protocol must take place at a given time, no time would be available for the conspirators to communicate. In the Chess Grandmaster Problem, if every move in each game must be made as a clock strikes one minute, then Alice will have no time to run from room to room. In the Mafia story, Bob and Carol will have no time to pass questions and answers to one another.

There is considerable mathematical difference between proofs that are only minimum-disclosure and those that are zero-knowledge. That distinction is beyond the scope of this book, but more sophisticated readers are welcome to peruse the references. The concepts were introduced in [626,619,622]. Further elaboration on their ideas, based on different mathematical assumptions, were developed in [240,319,239].

There are also different kinds of zero-knowledge proofs:

— **Perfect**. There is a simulator that gives transcripts identically distributed to real transcripts (the Hamiltonian cycle and graph isomorphism examples).
— **Statistical**. There is a simulator that gives transcripts identically distributed to real transcripts, except for some constant number of exceptions.
— **Computational**. There is a simulator that gives transcripts indistinguishable from real transcripts.
— **No-use**. A simulator may not exist, but we can prove that Victor will not learn any polynomial amount of information from the proof (the parallel example).

Over the years, extensive work, both theoretical and applied, has been done on minimum-disclosure and zero-knowledge proofs. Mike Burmester and Yvo Desmedt invented **broadcast interactive proofs**, where one prover can broadcast a zero-knowledge interactive proof to a large group of verifiers [280]. Cryptographers proved that *everything* that can be proven with an interactive proof can also be proven with a zero-knowledge interactive proof [753,137].

A good survey article on the topic is [548]. For additional mathematical details, variations, protocols, and applications, consult
[590,619,240,319,620,113,241,1528,660,238,591,617,510,592,214,104,216,832,
97,939,622,482,615,618,215,476,71]. *A lot* has been written on this subject.

5.2 Zero-Knowledge Proofs of Identity

In the real world, we often use physical tokens as proofs of identity: passports, driver's licenses, credit cards, and so on. The token contains something that links it to a person: a picture, usually, or a signature, but it could almost as easily be a thumbprint, a retinal scan, or a dental x-ray. Wouldn't it be nice to do the same thing digitally?

Using zero-knowledge proofs as proofs of identity was first proposed by Uriel Feige, Amos Fiat, and Adi Shamir [566,567]. Alice's private key becomes a function of her "identity." Using a zero-knowledge proof, she proves that she knows her private key and therefore proves her identity. Algorithms for this can be found in Section 23.11.

This idea is quite powerful. It allows a person to prove his identity without any physical token. However, it's not perfect. Here are some abuses.

The Chess Grandmaster Problem

Here's how Alice, who doesn't even know the rules to chess, can defeat a grandmaster. (This is sometimes called the Chess Grandmaster Problem.) She challenges both Gary Kasparov and Anatoly Karpov to a game, at the same time and place, but in separate rooms. She plays white against Kasparov and black against Karpov. Neither grandmaster knows about the other.

Karpov, as white, makes his first move. Alice records the move and walks into the room with Kasparov. Playing white, she makes the same move against Kasparov. Kasparov makes his first move as black. Alice records the move, walks into the room with Karpov, and makes the same move. This continues, until she wins one game and loses the other, or both games end in a draw.

(5) Peggy publishes all the commitments from step (2) as well as the solutions in step (4).
(6) Victor or Carol or whoever else is interested, verifies that steps (1) through (5) were executed properly.

This is amazing: Peggy can publish some data that contains no information about her secret, but can be used to convince anyone of the secret's existence. The protocol can also be used for digital signature schemes, if the challenge is set as a one-way hash of both the initial messages and the message to be signed.

This works because the one-way hash function acts as an unbiased random-bit generator. For Peggy to cheat, she has to be able to predict the output of the one-way hash function. (Remember, if she doesn't know the solution to the hard problem, she can do either (a) or (b) of step (4), but not both.) If she somehow knew what the one-way hash function would ask her to do, then she could cheat. However, there is no way for Peggy to force the one-way function to produce certain bits or to guess which bits it will produce. The one-way function is, in effect, Victor's surrogate in the protocol—randomly choosing one of two proofs in step (4).

In a noninteractive protocol, there must be many more iterations of the challenge/reply sequence. Peggy, not Victor, picks the hard problems using random numbers. She can pick different problems, hence different commitment vectors, till the hash function produces something she likes. In an interactive protocol, 10 iterations—a probability of 1 in 2^{10} (1 in 1024) that Peggy can cheat—may be fine. However, that's not enough for noninteractive zero-knowledge proofs. Remember that Mallory can always do either (a) or (b) of step (4). He can try to guess which he will be asked to do, go through steps (1) through (3), and see if he guessed right. If he didn't, he can try again—repeatedly. Making 1024 guesses is easy on a computer. To prevent this brute-force attack, noninteractive protocols need 64 iterations, or even 128 iterations, to be valid.

This is the whole point of using a one-way hash function: Peggy cannot predict the output of the hash function because she cannot predict its input. The commitments which are used as the input are only known after she solves the new problems.

Generalities

Blum proved that any mathematical theorem can be converted into a graph such that the proof of that theorem is equivalent to proving a Hamiltonian cycle in the graph. The general case that any **NP** statement has a zero-knowledge proof, assuming one-way functions and therefore good encryption algorithms, was proved in [620]. Any mathematical proof can be converted into a zero-knowledge proof. Using this technique, a researcher can prove to the world that he knows the proof of a particular theorem without revealing what that solution is. Blum could have published these results without revealing them.

There are also **minimum-disclosure proofs** [590]. In a minimum-disclosure proof, the following properties hold:

1. Peggy cannot cheat Victor. If Peggy does not know the proof, her chances of convincing Victor that she knows the proof are negligible.
2. Victor cannot cheat Peggy. He doesn't get the slightest hint of the proof, apart from the fact that Peggy knows the proof. In particular, Victor cannot demonstrate the proof to anyone else without proving it himself from scratch.

Zero-knowledge proofs have an additional condition:

3. Victor learns nothing from Peggy that he could not learn by himself without Peggy, apart from the fact that Peggy knows the proof.

The best she can do is to create a graph that is either isomorphic to *G* or one that has the same number of points and lines and a valid Hamiltonian cycle. While she has a 50 percent chance of guessing which proof Victor will ask her to perform in step (3), Victor can repeat the protocol enough times to convince himself that Peggy knows a Hamiltonian cycle for *G*.

Parallel Zero-Knowledge Proofs

The basic zero-knowledge protocol involves *n* exchanges between Peggy and Victor. Why not do them all in parallel:

(1) Peggy uses her information and *n* random numbers to transform the hard problem into *n* different isomorphic problems. She then uses her information and the random numbers to solve the *n* new hard problems.
(2) Peggy commits to the solution of the *n* new hard problems.
(3) Peggy reveals to Victor the *n* new hard problems. Victor cannot use these new problems to get any information about the original problems or its solutions.
(4) For each of the *n* new hard problems, Victor asks Peggy either to:
  (a) prove to him that the old and new problems are isomorphic, or
  (b) open the solution she committed to in step (2) and prove that it is a solution to the new problem.
(5) Peggy complies for each of the *n* new hard problems.

Unfortunately, it's not that simple. This protocol does not have the same zero-knowledge properties as the previous protocol. In step (4), Victor can choose the challenges as a one-way hash of all the values committed to in the second step, thus making the transcript nonsimulatable. It is still zero-knowledge, but of a different sort. It seems to be secure in practice, but no one knows how to prove it. We do know that in certain circumstances, certain protocols for certain problems can be run in parallel while retaining their zero-knowledge property [247,106,546,616].

Noninteractive Zero-Knowledge Proofs

Carol can't be convinced because the protocol is interactive, and she is not involved in the interaction. To convince Carol, and anyone else who may be interested, we need a noninteractive protocol.

Protocols have been invented for noninteractive zero-knowledge proofs [477,198,478,197]. These protocols do not require any interaction; Peggy could publish them and thereby prove to anyone who takes the time to check that the proof is valid.

The basic protocol is similar to the parallel zero-knowledge proof, but a one-way hash function takes the place of Victor:

(1) Peggy uses her information and *n* random numbers to transform the hard problem into *n* different isomorphic problems. She then uses her information and the random numbers to solve the *n* new hard problems.
(2) Peggy commits to the solution of the *n* new hard problems.
(3) Peggy uses all of these commitments together as a single input to a one-way hash function. (After all, the commitments are nothing more than bit strings.) She then saves the first *n* bits of the output of this one-way hash function.
(4) Peggy takes the *n* bits generated in step (3). For each *i*th new hard problem in turn, she takes the *i*th bit of those *n* bits and:
  (a) if it is a 0, she proves that the old and new problems are isomorphic, or
  (b) if it is a 1, she opens the solution she committed to in step (2) and proves that it is a solution to the new problem.

or

(b) proves that H and G_2 are isomorphic, without proving that H and G_1 are isomorphic.

(5) Peggy and Victor repeat steps (1) through (4) n times.

If Peggy does not know an isomorphism between G_1 and G_2, she cannot create graph H which is isomorphic to both. She can create a graph that is either isomorphic to G_1 or one that is isomorphic to G_2. Like the previous example, she has only a 50 percent chance of guessing which proof Victor will ask her to perform in step (3).

This protocol doesn't give Victor any useful information to aid him in figuring out an isomorphism between G_1 and G_2. Because Peggy generates a new graph H for each round of the protocol, he can get no information no matter how many rounds they go through the protocol. He won't be able to figure out an isomorphism between G_1 and G_2 from Peggy's answers.

In each round, Victor receives a new random permutation of H, along with an isomorphism between H and either G_1 or G_2. Victor could just as well have generated this by himself. Because Victor can create a simulation of the protocol, it can be proven to be zero-knowledge.

Hamiltonian Cycles

A variant of this example was first presented by Manuel Blum [196]. Peggy knows a circular, continuous path along the lines of a graph that passes through each point exactly once. This is called a **Hamiltonian cycle**. Finding a Hamiltonian cycle is another hard problem. Peggy has this piece of information—she probably got it by creating the graph with a certain Hamiltonian cycle—and this is what she wants to convince Victor that she knows.

Peggy knows the Hamiltonian cycle of a graph, G. Victor knows G, but not the Hamiltonian cycle. Peggy wants to prove to Victor that she knows this Hamiltonian cycle without revealing it. This is how she does it:

(1) Peggy randomly permutes G. She moves the points around and changes their labels to make a new graph, H. Since G and H are topologically isomorphic (i.e., the same graph), if she knows the Hamiltonian cycle of G then she can easily find the Hamiltonian cycle of H. If she didn't create H herself, determining the isomorphism between two graphs would be another hard problem; it could also take centuries of computer time. She then encrypts H to get H'. (This has to be a probabilistic encryption of each line in H, that is, an encrypted 0 or an encrypted 1 for each line in H.)

(2) Peggy gives Victor a copy of H'.

(3) Victor asks Peggy either to:

(a) prove to him that H' is an encryption of an isomorphic copy of G, or

(b) show him a Hamiltonian cycle for H.

(4) Peggy complies. She either:

(a) proves that H' is an encryption of an isomorphic copy of G by revealing the permutations and decrypting everything, without showing a Hamiltonian cycle for either G or H, or

(b) shows a Hamiltonian cycle for H by decrypting only those lines that constitute a Hamiltonian cycle, without proving that G and H are topologically isomorphic.

(5) Peggy and Victor repeat steps (1) through (4) n times.

If Peggy is honest, she can provide either proof in step (4) to Victor. However, if she does not know a Hamiltonian cycle for G, she cannot create an encrypted graph H' which can meet both challenges.

and out the other; there's no need for any cut-and-choose protocol. However, mathematical zero knowledge requires it.)

Assume that Peggy knows some information, and furthermore that the information is the solution to a hard problem. The basic zero-knowledge protocol consists of several rounds.

(1) Peggy uses her information and a random number to transform the hard problem into another hard problem, one that is isomorphic to the original problem. She then uses her information and the random number to solve this new instance of the hard problem.
(2) Peggy commits to the solution of the new instance, using a bit-commitment scheme.
(3) Peggy reveals to Victor the new instance. Victor cannot use this new problem to get any information about the original instance or its solution.
(4) Victor asks Peggy either to:
  (a) prove to him that the old and new instances are isomorphic (i.e., two different solutions to two related problems), or
  (b) open the solution she committed to in step (2) and prove that it is a solution to the new instance.
(5) Peggy complies.
(6) Peggy and Victor repeat steps (1) through (5) *n* times.

Remember the camcorder in the cave protocol? You can do the same thing here. Victor can make a transcript of the exchange between him and Peggy. He cannot use this transcript to convince Carol, because he can always collude with Peggy to build a simulator that fakes Peggy's knowledge. This argument can be used to prove that the proof is zero-knowledge.

The mathematics behind this type of proof is complicated. The problems and the random transformation must be chosen carefully, so that Victor does not get any information about the solution to the original problem, even after many iterations of the protocol. Not all hard problems can be used for zero-knowledge proofs, but a lot of them can.

Graph Isomorphism

An example might go a long way to explain this concept; this one comes from graph theory [619,622]. A graph is a network of lines connecting different points. If two graphs are identical except for the names of the points, they are called **isomorphic**. For an extremely large graph, finding whether two graphs are isomorphic can take centuries of computer time; it's one of those **NP-complete** problems discussed in Section 11.1.

Assume that Peggy knows the isomorphism between the two graphs, G_1 and G_2. The following protocol will convince Victor of Peggy's knowledge:

(1) Peggy randomly permutes G_1 to produce another graph, *H*, that is isomorphic to G_1. Because Peggy knows the isomorphism between *H* and G_1, she also knows the isomorphism between *H* and G_2. For anyone else, finding an isomorphism between G_1 and *H* or between G_2 and *H* is just as hard as finding an isomorphism between G_1 and G_2.
(2) Peggy sends *H* to Victor.
(3) Victor asks Peggy either to:
  (a) prove that *H* and G_1 are isomorphic, or
  (b) prove that *H* and G_2 are isomorphic.
(4) Peggy complies. She either:
  (a) proves that *H* and G_1 are isomorphic, without proving that *H* and G_2 are isomorphic,

(3) After Peggy has disappeared into the cave, Victor walks to point B.
(4) Victor shouts to Peggy, asking her either to:
  (a) come out of the left passage or
  (b) come out of the right passage.
(5) Peggy complies, using the magic words to open the secret door if she has to.
(6) Peggy and Victor repeat steps (1) through (5) *n* times.

Assume that Victor has a camcorder and records everything he sees. He records Peggy disappearing into the cave, he records when he shouts out where he wants Peggy to come out from, and he records Peggy coming out. He records all *n* trials. If he showed this recording to Carol, would she believe that Peggy knew the magic words to open the door? No. What if Peggy and Victor had agreed beforehand what Victor would call out, and Peggy would make sure that she went into that path. Then she could come out where Victor asked her every time, without knowing the magic words. Or maybe they couldn't do that. Peggy would go into one of the passages and Victor would call out a random request. If Victor guessed right, great; if he didn't, they would edit that trial out of the camcorder recording. Either way, Victor can get a recording showing exactly the same sequence of events as in a real proof where Peggy knew the magic words.

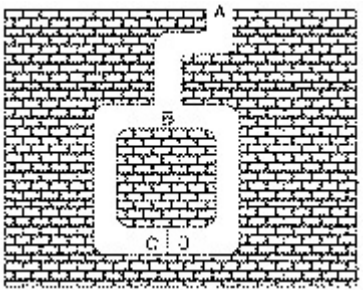

Figure 5.1 *The zero-knowledge cave.*

This shows two things. One, it is impossible for Victor to convince a third party of the proof's validity. And two, it proves that the protocol is zero-knowledge. In the case where Peggy did not know the magic words, Victor will obviously not learn anything from watching the recording. But since there is no way to distinguish a real recording from a faked recording, Victor cannot learn anything from the real proof—it must be zero knowledge.

The technique used in this protocol is called **cut and choose**, because of its similarity to the classic protocol for dividing anything fairly:

(1) Alice cuts the thing in half.
(2) Bob chooses one of the halves for himself.
(3) Alice takes the remaining half.

It is in Alice's best interest to divide fairly in step (1), because Bob will choose whichever half he wants in step (2). Michael Rabin was the first person to use the cut-and-choose technique in cryptography [1282]. The concepts of **interactive protocol** and zero-knowledge were formalized later [626,627].

The cut-and-choose protocol works because there is no way Peggy can repeatedly guess which side Victor will ask her to come out of. If Peggy doesn't know the secret, she can only come out the way she came in. She has a 50 percent chance of guessing which side Victor will ask in each round (sometimes called an **accreditation**) of the protocol, so she has a 50 percent chance of fooling him. The chance of her fooling him in two rounds is 25 percent, and the chance of her fooling him all *n* times is 1 in 2^n. After 16 rounds, Peggy has a 1 in 65,536 chance of fooling Victor. Victor can safely assume that if all 16 of Peggy's proofs are valid, then she must know the secret words to open the door between points C and D. (The cave analogy isn't perfect. Peggy can simply walk in one side

Chapter 5
Advanced Protocols

5.1 Zero-Knowledge Proofs

Here's another story:

> Alice: "I know the password to the Federal Reserve System computer, the ingredients in McDonald's secret sauce, and the contents of Volume 4 of Knuth."
>
> Bob: "No, you don't."
>
> Alice: "Yes, I do."
>
> Bob: "Do not!"
>
> Alice: "Do too!"
>
> Bob: "Prove it!"
>
> Alice: "All right. I'll tell you." She whispers in Bob's ear.
>
> Bob: "That's interesting. Now I know it, too. I'm going to tell *The Washington Post*."
>
> Alice: "Oops."

Unfortunately, the usual way for Alice to prove something to Bob is for Alice to tell him. But then he knows it, too. Bob can then tell anyone else he wants to and Alice can do nothing about it. (In the literature, different characters are often used in these protocols. Peggy is usually cast as the prover and Victor is the verifier. These names appear in the upcoming examples, instead of Alice and Bob.)

Using one-way functions, Peggy could perform a **zero-knowledge proof** [626]. This protocol proves to Victor that Peggy does have a piece of information, but it does not give Victor any way to know what the information is.

These proofs take the form of interactive protocols. Victor asks Peggy a series of questions. If Peggy knows the secret, she can answer all the questions correctly. If she does not, she has some chance—50 percent in the following examples—of answering correctly. After 10 or so questions, Victor will be convinced that Peggy knows the secret. Yet none of the questions or answers gives Victor any information about Peggy's information—only about her knowledge of it.

Basic Zero-Knowledge Protocol

Jean-Jacques Quisquater and Louis Guillou explain zero-knowledge with a story about a cave [1281]. The cave, illustrated in Figure 5.1, has a secret. Someone who knows the magic words can open the secret door between C and D. To everyone else, both passages lead to dead ends.

Peggy knows the secret of the cave. She wants to prove her knowledge to Victor, but she doesn't want to reveal the magic words. Here's how she convinces him:

(1) Victor stands at point A.
(2) Peggy walks all the way into the cave, either to point C or point D.

a bill, which it trusts the bank to pay. The first bank passes the bill on to the customer, which it trusts the customer to pay. When the customer pays, the first bank transfers additional funds to the second bank. All transactions are handled through an intermediary, which acts sort of like an electronic Federal Reserve: settling accounts among banks, logging messages, and creating an audit trail.

Exchanges between the customer, merchant, and various banks are outlined in [988]. Unless everyone colludes against the customer, his anonymity is assured. However, this is not digital cash; it is easy for the bank to cheat. The protocol allows customers to keep the advantages of credit cards without giving up their privacy.

called an **observer** [332, 341, 387]. The observer chip keeps a mini database of all the pieces of digital cash spent by that smart card. If Alice attempts to copy some digital cash and spend it twice, the imbedded observer chip would detect the attempt and would not allow the transaction. Since the observer chip is tamperproof, Alice cannot erase the mini-database without permanently damaging the smart card. The cash can wend its way through the economy; when it is finally deposited, the bank can examine the cash and determine who, if anyone, cheated.

Digital cash protocols can also be divided along another line. **Electronic coins** have a fixed value; people using this system will need several coins in different denominations. **Electronic checks** can be used for any amount up to a maximum value and then returned for a refund of the unspent portion.

Two excellent and completely different off-line electronic coin protocols are [225, 226, 227] and [563, 564, 565]. A system called NetCash, with weaker anonymity properties, has also been proposed [1048, 1049]. Another new system is [289].

In [1211], Tatsuaki Okamoto and Kazuo Ohta list six properties of an ideal digital cash system:

1. Independence. The security of the digital cash is not dependent on any physical location. The cash can be transferred through computer networks.
2. Security. The digital cash cannot be copied and reused.
3. Privacy (Untraceability). The privacy of the user is protected; no one can trace the relationship between the user and his purchases.
4. Off-line Payment. When a user pays for a purchase with electronic cash, the protocol between the user and the merchant is executed off-line. That is, the shop does not need to be linked to a host to process the user's payment.
5. Transferability. The digital cash can be transferred to other users.
6. Divisibility. A piece of digital cash in a given amount can be subdivided into smaller pieces of cash in smaller amounts. (Of course, everything has to total up properly in the end.)

The protocols previously discussed satisfy properties 1, 2, 3, and 4, but not 5 and 6. Some on-line digital cash systems satisfy all properties except 4 [318, 413, 1243]. The first off-line digital cash system that satisfies properties 1, 2, 3, and 4, similar to the one just discussed, was proposed in [339]. Okamoto and Ohta proposed a system that satisfies properties 1 through 5 [1209]; they also proposed a system that satisfies properties 1 through 6 as well, but the data requirement for a single purchase is approximately 200 megabytes. Another off-line divisible coin system is described in [522].

The digital cash scheme proposed in [1211], by the same authors, satisfies properties 1 through 6, without the enormous data requirements. The total data transfer for a payment is about 20 kilobytes, and the protocol can be completed in several seconds. The authors consider this the first ideal untraceable electronic cash system.

Anonymous Credit Cards

This protocol [988] uses several different banks to protect the identity of the customer. Each customer has an account at two different banks. The first bank knows the person's identity and is willing to extend him credit. The second bank knows the customer only under a pseudonym (similar to a numbered Swiss bank account).

The customer can withdraw funds from the second bank by proving that the account is his. However, the bank does not know the person and is unwilling to extend him credit. The first bank knows the customer and transfers funds to the second bank—without knowing the pseudonym. The customer then spends these funds anonymously. At the end of the month, the second bank gives the first bank

cheater. If Eve steals and spends Alice's cash before Alice can, then Alice will be identified as a cheater. There's no way to prevent this; it is a direct result of the anonynimity of the cash. Both Alice and the merchant have to protect their bits as they would paper money.

This protocol lies somewhere between an arbitrated protocol and a self-enforcing protocol. Both Alice and the merchant trust the bank to make good on the money orders, but Alice does not have to trust the bank with knowledge of her purchases.

Digital Cash and the Perfect Crime

Digital cash has its dark side, too. Sometimes people don't want so much privacy. Watch Alice commit the perfect crime [1575]:

(1) Alice kidnaps a baby.
(2) Alice prepares 10, 000 anonymous money orders for $1000 (or as many as she wants for whatever denomination she wants).
(3) Alice blinds all 10, 000 money orders, using a blind signature protocol. She sends them to the authorities with the threat to kill the baby unless the following instructions are met:
  (a) Have a bank sign all 10, 000 money orders.
  (b) Publish the results in a newspaper.
(4) The authorities comply.
(5) Alice buys a newspaper, unblinds the money orders, and starts spending them. There is no way for the authorities to trace the money orders to her.
(6) Alice frees the baby.

Note that this situation is much worse than any involving physical tokens—cash, for example. Without physical contact, the police have less opportunity to apprehend the kidnapper.

In general, though, digital cash isn't a good deal for criminals. The problem is that the anonymity only works one way: The spender is anonymous, but the merchant is not. Moreover, the merchant cannot hide the fact that he received money. Digital cash will make it easy for the government to determine how much money you made, but impossible to determine what you spent it on.

Practical Digital Cash

A Dutch company, DigiCash, owns most of the digital cash patents and has implemented digital cash protocols in working products. Anyone interested should contact DigiCash BV, Kruislaan 419, 1098 VA Amsterdam, Netherlands.

Other Digital Cash Protocols

There are other digital cash protocols; see [707, 1554, 734, 1633, 973]. Some of them involve some pretty complicated mathematics. Generally, the various digital cash protocols can be divided into various categories. **On-line** systems require the merchant to communicate with the bank at every sale, much like today's credit-card protocols. If there is a problem, the bank doesn't accept the cash and Alice cannot cheat.

Off-line systems, like Protocol #4, require no communication between the merchant and the bank until after the transaction between the merchant and the customer. These systems do not prevent Alice from cheating, but instead detect her cheating. Protocol #4 detected her cheating by making Alice's identity known if she tried to cheat. Alice knows that this will happen, so she doesn't cheat.

Another way is to create a special smart card (see Section 24.13) containing a tamperproof chip

it is the same, the bank knows that the merchant copied the money order. If it is different, the bank knows that the person who bought the money order photocopied it. Since the second merchant who accepted the money order handed Alice a different selector string than did the first merchant, the bank finds a bit position where one merchant had Alice open the left half and the other merchant had Alice open the right half. The bank XORs the two halves together to reveal Alice's identity.

This is quite an amazing protocol, so let's look at it from various angles.

Can Alice cheat? Her digital money order is nothing more than a string of bits, so she can copy it. Spending it the first time won't be a problem; she'll just complete the protocol and everything will go smoothly. The merchant will give her a random *n*-bit selector string in step (7) and Alice will open either the left half or right half of each I_i in step (8). In step (10), the bank will record all of this data, as well as the money order's uniqueness string.

When she tries to use the same digital money order a second time, the merchant (either the same merchant or a different merchant) will give her a different random selector string in step (7). Alice must comply in step (8); not doing so will immediately alert the merchant that something is suspicious. Now, when the merchant brings the money order to the bank in step (10), the bank would immediately notice that a money order with the same uniqueness string was already deposited. The bank then compares the opened halves of the identity strings. The odds that the two random selector strings are the same is 1 in 2^n; it isn't likely to happen before the next ice age. Now, the bank finds a pair with one half opened the first time and the other half opened the second time. It XORs the two halves together, and out pops Alice's name. The bank knows who tried to spend the money order twice.

Note that this protocol doesn't keep Alice from trying to cheat; it detects her cheating with almost certainty. Alice can't prevent her identity from being revealed if she cheats. She can't change either the uniqueness string or any of the identity strings, because then the bank's signature will no longer be valid. The merchant will immediately notice that in step (6).

Alice could try to sneak a bad money order past the bank, one on which the identity strings don't reveal her name; or better yet, one whose identity strings reveal someone else's name. The odds of her getting this ruse past the bank in step (3) are 1 in n . These aren't impossible odds, but if you make the penalty severe enough, Alice won't try it. Or, you could increase the number of redundant money orders that Alice makes in step (1).

Can the merchant cheat? His chances are even worse. He can't deposit the money order twice; the bank will notice the repeated use of the selector string. He can't fake blaming Alice; only she can open any of the identity strings.

Even collusion between Alice and the merchant can't cheat the bank. As long as the bank signs the money order with the uniqueness string, the bank is assured of only having to make good on the money order once.

What about the bank? Can it figure out that the money order it accepted from the merchant was the one it signed for Alice? Alice is protected by the blind signature protocol in steps (2) through (5). The bank cannot make the connection, even if it keeps complete records of every transaction. Even more strongly, there is no way for the bank and the merchant to get together to figure out who Alice is. Alice can walk in the store and, completely anonymously, make her purchase.

Eve can cheat. If she can eavesdrop on the communication between Alice and the merchant, and if she can get to the bank before the merchant does, she can deposit the digital cash first. The bank will accept it and, even worse, when the merchant tries to deposit the cash he will be identified as a

giving the same identity string in step (7). Unless the merchant keeps a database of money orders it already received, he would be fooled. The next protocol eliminates that problem.

Protocol #4

If it turns out that the person who bought the money order tried to cheat the merchant, the bank would want to know who that person was. To do that requires moving away from a physical analogy and into the world of cryptography.

The technique of secret splitting can be used to hide Alice's name in the digital money order.

(1) Alice prepares *n* anonymous money orders for a given amount.
Each of the money orders contains a different random uniqueness string, *X*, one long enough to make the chance of two being identical negligible.
On each money order, there are also *n* pairs of identity bit strings, I_1, I_2,..., I_n. (Yes, that's *n* different pairs on *each* check.) Each of these pairs is generated as follows: Alice creates a string that gives her name, address, and any other piece of identifying information that the bank wants to see. Then, she splits it into two pieces using the secret splitting protocol (see Section 3.6). Then, she commits to each piece using a bit-commitment protocol.
For example, I_{37} consists of two parts: I_{37L} and I_{37R}. Each part is a bit-committed packet that Alice can be asked to open and whose proper opening can be instantly verified. Any pair (e.g., I_{37L} and I_{37R}, but not I_{37L} and I_{38R}), reveals Alice's identity.
Each of the money orders looks like this:

```
   Amount
   Uniqueness String: X
Identity Strings:  I1 = (I1L,I1R)
                   I2 =       (I2L,I2R)
                    ....
                   In = (InL,  InR)
```

(2) Alice blinds all *n* money orders, using a blind signature protocol. She gives them all to the bank.
(3) The bank asks Alice to unblind *n* - 1 of the money orders at random and confirms that they are all well formed. The bank checks the amount, the uniqueness string, and asks Alice to reveal all of the identity strings.
(4) If the bank is satisfied that Alice did not make any attempts to cheat, it signs the one remaining blinded money order. The bank hands the blinded money order back to Alice and deducts the amount from her account.
(5) Alice unblinds the money order and spends it with a merchant.
(6) The merchant verifies the bank's signature to make sure the money order is legitimate.
(7) The merchant asks Alice to randomly reveal either the left half or the right half of each identity string on the money order. In effect, the merchant gives Alice a random *n*- bit **selector string**, b_1, b_2,..., b_n. Alice opens either the left or right half of I_i, depending on whether b_i is a 0 or a 1.
(8) Alice complies.
(9) The merchant takes the money order to the bank.
(10) The bank verifies the signature and checks its database to make sure a money order with the same uniqueness string has not been previously deposited. If it hasn't, the bank credits the amount to the merchant's account. The bank records the uniqueness string and all of the identity information in a database.
(11) If the uniqueness string is in the database, the bank refuses to accept the money order. Then, it compares the identity string on the money order with the one stored in the database. If

carbon paper to the money order. The bank hands the unopened envelope back to Alice and deducts $1000 from her account.
(5) Alice opens the envelope and spends the money order with a merchant.
(6) The merchant checks for the bank's signature to make sure the money order is legitimate.
(7) The merchant takes the money order to the bank.
(8) The bank verifies its signature and checks its database to make sure a money order with the same uniqueness string has not been previously deposited. If it hasn't, the bank credits $1000 to the merchant's account. The bank records the uniqueness string in a database.
(9) If it has been previously deposited, the bank doesn't accept the money order.

Now, if Alice tries to spend a photocopy of the money order, or if the merchant tries to deposit a photocopy of the money order, the bank will know about it.

Protocol #3

The previous protocol protects the bank from cheaters, but it doesn't identify them. The bank doesn't know if the person who bought the money order (the bank has no idea it's Alice) tried to cheat the merchant or if the merchant tried to cheat the bank. This protocol corrects that:

(1) Alice prepares 100 anonymous money orders for $1000 each. On each of the money orders she includes a different random uniqueness string, one long enough to make the chance of another person also using it negligible.
(2) Alice puts one each, and a piece of carbon paper, into 100 different envelopes. She gives them all to the bank.
(3) The bank opens 99 envelopes and confirms that each is a money order for $1000 and that all the random strings are different.
(4) The bank signs the one remaining unopened envelope. The signature goes through the carbon paper to the money order. The bank hands the unopened envelope back to Alice and deducts $1000 from her account.
(5) Alice opens the envelope and spends the money order with a merchant.
(6) The merchant checks for the bank's signature to make sure the money order is legitimate.
(7) The merchant asks Alice to write a random identity string on the money order.
(8) Alice complies.
(9) The merchant takes the money order to the bank.
(10) The bank verifies the signature and checks its database to make sure a money order with the same uniqueness string has not been previously deposited. If it hasn't, the bank credits $1000 to the merchant's account. The bank records the uniqueness string and the identity string in a database.
(11) If the uniqueness string is in the database, the bank refuses to accept the money order. Then, it compares the identity string on the money order with the one stored in the database. If it is the same, the bank knows that the merchant photocopied the money order. If it is different, the bank knows that the person who bought the money order photocopied it.

This protocol assumes that the merchant cannot change the identity string once Alice writes it on the money order. The money order might have a series of little squares, which the merchant would require Alice to fill in with either Xs or Os. The money order might be made out of paper that tears if erased.

Since the interaction between the merchant and the bank takes place after Alice spends the money, the merchant could be stuck with a bad money order. Practical implementations of this protocol might require Alice to wait near the cash register during the merchant-bank interaction, much the same way as credit-card purchases are handled today.

Alice could also frame the merchant. She could spend a copy of the money order a second time,

same piece of digital cash into two different accounts, he will be detected—but Alice will remain anonymous. Sometimes this is called **anonymous digital cash** to differentiate it from digital money with an audit trail, such as credit cards.

A great social need exists for this kind of thing. With the growing use of the Internet for commercial transactions, there is more call for network-based privacy and anonymity in business. (There are good reasons people are reluctant to send their credit card numbers over the Internet.) On the other hand, banks and governments seem unwilling to give up the control that the current banking system's audit trail provides. They'll have to, though. All it will take for digital cash to catch on is for some trustworthy institution to be willing to convert the digits to real money.

Digital cash protocols are very complex. We'll build up to one, a step at a time. For more formal details, read [318, 339, 325, 335, 340]. Realize that this is just one digital cash protocol; there are others.

Protocol #1

The first few protocols are physical analogies of cryptographic protocols. This first protocol is a simplified physical protocol for anonymous money orders:

(1) Alice prepares 100 anonymous money orders for $1000 each.
(2) Alice puts one each, and a piece of carbon paper, into 100 different envelopes. She gives them all to the bank.
(3) The bank opens 99 envelopes and confirms that each is a money order for $1000.
(4) The bank signs the one remaining unopened envelope. The signature goes through the carbon paper to the money order. The bank hands the unopened envelope back to Alice, and deducts $1000 from her account.
(5) Alice opens the envelope and spends the money order with a merchant.
(6) The merchant checks for the bank's signature to make sure the money order is legitimate.
(7) The merchant takes the money order to the bank.
(8) The bank verifies its signature and credits $1000 to the merchant's account.

This protocol works. The bank never sees the money order it signed, so when the merchant brings it to the bank, the bank has no idea that it was Alice's. The bank is convinced that it is valid, though, because of the signature. The bank is confident that the unopened money order is for $1000 (and not for $100, 000 or $100, 000, 000) because of the cut-and-choose protocol (see Section 5.1). It verifies the other 99 envelopes, so Alice has only a 1 percent chance of cheating the bank. Of course, the bank will make the penalty for cheating great enough so that it isn't worth that chance. If the bank refuses to sign the last check (if Alice is caught cheating) without penalizing Alice, she will continue to try until she gets lucky. Prison terms are a better deterrent.

Protocol #2

The previous protocol prevents Alice from writing a money order for more than she claims to, but it doesn't prevent Alice from photocopying the money order and spending it twice. This is called the **double spending problem**; to solve it, we need a complication:

(1) Alice prepares 100 anonymous money orders for $1000 each. On each money order she includes a different random uniqueness string, one long enough to make the chance of another person also using it negligible.
(2) Alice puts one each, and a piece of carbon paper, into 100 different envelopes. She gives them all to the bank.
(3) The bank opens 99 envelopes and confirms that each is a money order for $1000.
(4) The bank signs the one remaining unopened envelope. The signature goes through the

(1) The users arrange themselves into a circle.
(2) At regular intervals, adjacent pairs of users flip coins between them, using some fair coin flip protocol secure from eavesdroppers.
(3) After every flip, each user announces either "same" or "different."

If Alice wishes to broadcast a message, she simply starts inverting her statement in those rounds corresponding to a 1 in the binary representation of her message. For example, if her message were "1001, " she would invert her statement, tell the truth, tell the truth, and then invert her statement. Assuming the result of her flips were "different, " "same, " "same, " "same, " she would say "same, " "same, " "same, " "different."

If Alice notices that the overall outcome of the protocol doesn't match the message she is trying to send, she knows that someone else is trying to send a message at the same time. She then stops sending the message and waits some random number of rounds before trying again. The exact parameters have to be worked out based on the amount of message traffic on this network, but the idea should be clear.

To make things even more interesting, these messages can be encrypted in another user's public keys. Then, when everyone receives the message (a real implementation of this should add some kind of standard message-beginning and message-ending strings), only the intended recipient can decrypt and read it. No one else knows who sent it. No one else knows who could read it. Traffic analysis, which traces and compiles patterns of people's communications even though the messages themselves may be encrypted, is useless.

An alternative to flipping coins between adjacent parties would be for them to keep a common file of random bits. Maybe they could keep them on a CD-ROM, or one member of the pair could generate a pile of them and send them to the other party (encrypted, of course). Alternatively, they could agree on a cryptographically secure pseudo-random-number generator between them, and they could each generate the same string of pseudo-random bits for the protocol.

One problem with this protocol is that while a malicious participant cannot read any messages, he can disrupt the system unobserved by lying in step (3). There is a modification to the previous protocol that detects disruption [1578, 1242]; the problem is called "The Dining Cryptographers in the Disco."

6.4 Digital Cash

Cash is a problem. It's annoying to carry, it spreads germs, and people can steal it from you. Checks and credit cards have reduced the amount of physical cash flowing through society, but the complete elimination of cash is virtually impossible. It'll never happen; drug dealers and politicians would never stand for it. Checks and credit cards have an audit trail; you can't hide to whom you gave money.

On the other hand, checks and credit cards allow people to invade your privacy to a degree never before imagined. You might never stand for the police following you your entire life, but the police can watch your financial transactions. They can see where you buy your gas, where you buy your food, who you call on the telephone—all without leaving their computer terminals. People need a way to protect their anonymity in order to protect their privacy.

Happily, there is a complicated protocol that allows for authenticated but untraceable messages. Lobbyist Alice can transfer **digital cash** to Congresscritter Bob so that newspaper reporter Eve does not know Alice's identity. Bob can then deposit that electronic money into his bank account, even though the bank has no idea who Alice is. But if Alice tries to buy cocaine with the same piece of digital cash she used to bribe Bob, she will be detected by the bank. And if Bob tries to deposit the

1. Alice can enter her input without Bob's being able to learn it.
2. Bob can enter his input without Alice's being able to learn it.
3. Both Alice and Bob can calculate the output, with both parties being sure the output is correct and that neither party has tampered with it.

Details on secure circuit evaluation can be found in [831].

6.3 Anonymous Message Broadcast

You can't go out to dinner with a bunch of cryptographers without raising a ruckus. In [321], David Chaum introduced the Dining Cryptographers Problem:

> Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maître d'hôtel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been the NSA. The three cryptographers respect each other's right to make an anonymous payment, but they wonder if the NSA is paying.

How do the cryptographers, named Alice, Bob, and Carol, determine if one of them is paying for dinner, while at the same time preserving the anonymity of the payer?

Chaum goes on to solve the problem:

> Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer to his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see—the one he flipped and the one his left-hand neighbor flipped—fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number of differences indicates that NSA is paying (assuming that the dinner was paid for only once). Yet, if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is.

To see that this works, imagine Alice trying to figure out which other cryptographer paid for dinner (assuming that neither she nor the NSA paid). If she sees two different coins, then either both of the other cryptographers, Bob and Carol, said, "same" or both said, "different." (Remember, an odd number of cryptographers saying "different" indicates that one of them paid.) If both said, "different, " then the payer is the cryptographer closest to the coin that is the same as the hidden coin (the one that Bob and Carol flipped). If both said, "same, " then the payer is the cryptographer closest to the coin that is different from the hidden coin. However, if Alice sees two coins that are the same, then either Bob said, "same" and Carol said, "different, " or Bob said, "different" and Carol said, "same." If the hidden coin is the same as the two coins she sees, then the cryptographer who said, "different" is the payer. If the hidden coin is different from the two coins she sees, then the cryptographer who said, "same" is the payer. In all of these cases, Alice needs to know the result of the coin flipped between Bob and Carol to determine which of them paid.

This protocol can be generalized to any number of cryptographers; they all sit in a ring and flip coins among them. Even two cryptographers can perform the protocol. Of course, they know who paid, but someone watching the protocol could tell only if one of the two paid or if the NSA paid; they could not tell which cryptographer paid.

The applications of this protocol go far beyond sitting around the dinner table. This is an example of **unconditional sender and recipient untraceability**. A group of users on a network can use this protocol to send anonymous messages.

together they will determine whether $a = b$, such that Bob does not learn anything additional about a and Alice does not learn anything additional about b. Details are in Section 23.14.

Protocol #4

This is another problem for secure multiparty computation [1373]: A council of seven meets regularly to cast secret ballots on certain issues. (All right, they rule the world—don't tell anyone I told you.) All council members can vote yes or no. In addition, two parties have the option of casting "super votes": *S*-yes and *S*-no. They do not have to cast super votes; they can cast regular votes if they prefer. If no one casts any super votes, then the majority of votes decides the issue. In the case of a single or two equivalent super votes, all regular votes are ignored. In the case of two contradicting super votes, the majority of regular votes decides. We want a protocol that securely performs this style of voting.

Two examples should illustrate the voting process. Assume there are five regular voters, N_1 through $N_{5,}$ and two super voters: S_1 and S_2. Here's the vote on issue #1:

S_1	S_2	N_1	N_2	N_3	N_4	N_5
S-yes	no	no	no	no	yes	yes

In this instance the only vote that matters is S_1 's, and the result is "yes."

Here is the vote on issue #2:

S_1	S_2	N_1	N_2	N_3	N_4	N_5
S-yes	S-no	no	no	no	yes	yes

Here the two super votes cancel and the majority of regular "no" votes decide the issue.

If it isn't important to hide the knowledge of whether the super vote or the regular vote was the deciding vote, this is an easy application of a secure voting protocol. Hiding that knowledge requires a more complicated secure multiparty computation protocol.

This kind of voting could occur in real life. It could be part of a corporation's organizational structure, where certain people have more power than others, or it could be part of the United Nations's procedures, where certain nations have more power than others.

Multiparty Unconditionally Secure Protocols

This is just a simple case of a general theorem: Any function of n inputs can be computed by a set of n players in a way that will let all learn the value of the function, but any set of less than $n/2$ players will not get any additional information that does not follow from their own inputs and the value of the output information. For details, see [136, 334, 1288, 621].

Secure Circuit Evaluation

Alice has her input, a. Bob has his input, b. Together they wish to compute some general function, $f(a,b)$, such that Alice learns nothing about Bob's input and Bob learns nothing about Alice's input. The general problem of secure multiparty computation is also called **secure circuit evaluation**. Here, Alice and Bob can create an arbitrary Boolean circuit. This circuit accepts inputs from Alice and from Bob and produces an output. Secure circuit evaluation is a protocol that accomplishes three things:

ability to handle situations more complex than determining which of two numbers is greater. And two, if Alice and Bob were really concerned about the secrecy of their information, they would be forced to drown the waiter in a bowl of vichyssoise, lest he tell the wine steward.

Public-key cryptography offers a far less violent solution. There is a protocol by which Alice, who knows a value *a*, and Bob, who knows a value *b*, can together determine if $a < b$, so that Alice gets no additional information about *b* and Bob gets no additional information about *a*. And, both Alice and Bob are convinced of the validity of the computation. Since the cryptographic algorithm used is an essential part of the protocol, details can be found in Section 23.14.

Of course, this protocol doesn't protect against active cheaters. There's nothing to stop Alice (or Bob, for that matter) from lying about her age. If Bob were a computer program that blindly executed the protocol, Alice could learn his age (is the age of a computer program the length of time since it was written or the length of time since it started running?) by repeatedly executing the protocol. Alice might give her age as 60. After learning that she is older, she could execute the protocol again with her age as 30. After learning that Bob is older, she could execute the protocol again with her age as 45, and so on, until Alice discovers Bob's age to any degree of accuracy she wishes.

Assuming that the participants don't actively cheat, it is easy to extend this protocol to multiple participants. Any number of people can find out the order of their ages by a sequence of honest applications of the protocol; and no participant can learn the age of another.

Protocol #3

Alice likes to do kinky things with teddy bears. Bob has erotic fantasies about marble tables. Both are pretty embarrassed by their particular fetish, but would love to find a mate who shared in their...um...lifestyle.

Here at the Secure Multiparty Computation Dating Service, we've designed a protocol for people like them. We've numbered an astonishing list of fetishes, from "aardvarks" to "zoot suits." Discreetly separated by a modem link, Alice and Bob can participate in a secure multiparty protocol. Together, they can determine whether they share the same fetish. If they do, they might look forward to a lifetime of bliss together. If they don't, they can part company secure in the knowledge that their particular fetish remains confidential. No one, not even the Secure Multiparty Computation Dating Service, will ever know.

Here's how it works:

(1) Using a one-way function, Alice hashes her fetish into a seven-digit string.
(2) Alice uses the seven-digit string as a telephone number, calls the number, and leaves a message for Bob. If no one answers or the number is not in service, Alice applies a one-way function to the telephone number until she finds someone who can play along with the protocol.
(3) Alice tells Bob how many times she had to apply the one-way hash function to her fetish.
(4) Bob hashes his fetish the same number of times that Alice did. He also uses the seven-digit string as a telephone number, and asks the person at the other end whether there were any messages for him.

Note that Bob has a chosen-plaintext attack. He can hash common fetishes and call the resulting telephone numbers, looking for messages for him. This protocol only really works if there are enough possible plaintext messages for this to be impractical.

There's also a mathematical protocol, one similar to Protocol #2. Alice knows *a*, Bob knows *b*, and

not practical for large-scale elections.

Another, more complex, voting protocol that solves some of these problems can be found in [770, 771]. There is even a voting protocol that uses multiple-key ciphers [219]. Yet another voting protocol, which claims to be practical for large-scale elections, is in [585]. And [347] allows voters to abstain.

Voting protocols work, but they make it easier to buy and sell votes. The incentives become considerably stronger as the buyer can be sure that the seller votes as promised. Some protocols are designed to be **receipt-free**, so that it is impossible for a voter to prove to someone else that he voted in a certain way [117, 1170, 1372].

6.2 Secure Multiparty Computation

Secure multiparty computation is a protocol in which a group of people can get together and compute any function of many variables in a special way. Each participant in the group provides one or more variables. The result of the function is known to everyone in the group, but no one learns anything about the inputs of any other members other than what is obvious from the output of the function. Here are some examples:

Protocol #1

How can a group of people calculate their average salary without anyone learning the salary of anyone else?

> **(1)** Alice adds a secret random number to her salary, encrypts the result with Bob's public key, and sends it to Bob.
> **(2)** Bob decrypts Alice's result with his private key. He adds his salary to what he received from Alice, encrypts the result with Carol's public key, and sends it to Carol.
> **(3)** Carol decrypts Bob's result with her private key. She adds her salary to what she received from Bob, encrypts the result with Dave's public key, and sends it to Dave.
> **(4)** Dave decrypts Carol's result with his private key. He adds his salary to what he received from Carol, encrypts the result with Alice's public key, and sends it to Alice.
> **(5)** Alice decrypts Dave's result with her private key. She subtracts the random number from step (1) to recover the sum of everyone's salaries.
> **(6)** Alice divides the result by the number of people (four, in this case) and announces the result.

This protocol assumes that everyone is honest; they may be curious, but they follow the protocol. If any participant lies about his salary, the average will be wrong. A more serious problem is that Alice can misrepresent the result to everyone. She can subtract any number she likes in step (5), and no one would be the wiser. Alice could be prevented from doing this by requiring her to commit to her random number using any of the bit-commitment schemes from Section 4.9, but when she revealed her random number at the end of the protocol Bob could learn her salary.

Protocol #2

Alice and Bob are at a restaurant together, having an argument over who is older. They don't, however, want to tell the other their age. They could each whisper their age into the ear of a trusted neutral party (the waiter, for example), who could compare the numbers in his head and announce the result to both Alice and Bob.

The above protocol has two problems. One, your average waiter doesn't have the computational

Substituting one vote for another during round one of the protocol is more subtle. Alice can't do it in step (3), because Bob, Carol, or Dave will detect it in step (5), (6), or (7). Bob could try in step (5). If he replaces Carol's or Dave's vote (remember, he doesn't know which vote corresponds to which voter), Carol or Dave will notice in step (6) or (7). They wouldn't know who tampered with their vote (although it would have had to be someone who had already handled the votes), but they would know that their vote was tampered with. If Bob is lucky and picks Alice's vote to replace, she won't notice until the second round. Then, she will notice her vote missing in step (8). Still, she would not know who tampered with her vote. In the first round, the votes are shuffled from one step to the other and unsigned; it is impossible for anyone to backtrack through the protocol to determine who tampered with the votes.

Another form of cheating is to try to figure out who voted for whom. Because of the scrambling in the first round, it is impossible for someone to backtrack through the protocol and link votes with voters. The removal of the random strings during the first round is also crucial to preserving anonymity. If they are not removed, the scrambling of the votes could be reversed by re-encrypting the emerging votes with the scrambler's public key. As the protocol stands, the confidentiality of the votes is secure.

Even more strongly, because of the initial random string, R_1, even identical votes are encrypted differently at every step of the protocol. No one knows the outcome of the vote until step (11).

What are the problems with this protocol? First, the protocol has an enormous amount of computation. The example described had only four voters and *it* was complicated. This would never work in a real election, with tens of thousands of voters. Second, Dave learns the results of the election before anyone else does. While he still can't affect the outcome, this gives him some power that the others do not have. On the other hand, this is also true with centralized voting schemes.

The third problem is that Alice can copy anyone else's vote, even though she does not know what it is beforehand. To see why this could be a problem, consider a three-person election between Alice, Bob, and Eve. Eve doesn't care about the result of the election, but she wants to know how Alice voted. So she copies Alice's vote, and the result of the election is guaranteed to be equal to Alice's vote.

Other Voting Schemes

Many complex secure election protocols have been proposed. They come in two basic flavors. There are mixing protocols, like "Voting without a Central Tabulating Facility, " where everyone's vote gets mixed up so that no one can associate a vote with a voter.

There are also divided protocols, where individual votes are divided up among different tabulating facilities such that no single one of them can cheat the voters [360, 359, 118, 115]. These protocols only protect the privacy of voters to the extent that different "parts" of the government (or whoever is administering the voting) do not conspire against the voter. (This idea of breaking a central authority into different parts, who are only trusted when together, comes from [316].)

One divided protocol is [1371]. The basic idea is that each voter breaks his vote into several shares. For example, if the vote were "yes" or "no, " a 1 could indicate "yes" and a 0 could indicate "no"; the voter would then generate several numbers whose sum was either 0 or 1. These shares are sent to tabulating facilities, one to each, and are also encrypted and posted. Each center tallies the shares it receives (there are protocols to verify that the tally is correct) and the final vote is the sum of all the tallies. There are also protocols to ensure that each voter's shares add up to 0 or 1.

Another protocol, by David Chaum [322], ensures that voters who attempt to disrupt the election can be traced. However, the election must then be restarted without the interfering voter; this approach is

Each vote now looks like this:

$E_C(R_3,E_D\ (R_2,E_A(E_B(E_C(E_D(V,R_1))))))$

(6) Carol decrypts all of the votes with her private key, checks to see that her vote is among the set of votes, removes all the random strings at that level, scrambles all the votes, and then sends the result to Dave.

Each vote now looks like this:

$E_D(R_2,E_A(E_B(E_C(E_D(V,R_1)))))$

(7) Dave decrypts all of the votes with his private key, checks to see that his vote is among the set of votes, removes all the random strings at that level, scrambles all the votes, and sends them to Alice.

Each vote now looks like this:

$E_A(E_B(E_C(E_D(V,R_1))))$

(8) Alice decrypts all the votes with her private key, checks to see that her vote is among the set of votes, signs all the votes, and then sends the result to Bob, Carol, and Dave.

Each vote now looks like this:

$S_A(E_B(E_C(E_D(V,R_1))))$

(9) Bob verifies and deletes Alice's signatures. He decrypts all the votes with his private key, checks to see that his vote is among the set of votes, signs all the votes, and then sends the result to Alice, Carol, and Dave.

Each vote now looks like this:

$S_B(E_C(E_D(V,R_1)))$

(10) Carol verifies and deletes Bob's signatures. She decrypts all the votes with her private key, checks to see that her vote is among the set of votes, signs all the votes, and then sends the result to Alice, Bob, and Dave.

Each vote now looks like this:

$S_C(E_D(V,R_1))$

(11) Dave verifies and deletes Carol's signatures. He decrypts all the votes with his private key, checks to see that his vote is among the set of votes, signs all the votes, and then sends the result to Alice, Bob, and Carol.

Each vote now looks like this:

$S_D(V,R_1)$

(12) All verify and delete Dave's signature. They check to make sure that their vote is among the set of votes (by looking for their random string among the votes).

(13) Everyone removes the random strings from each vote and tallies the votes.

Not only does this protocol work, it is also self-adjudicating. Alice, Bob, Carol, and Dave will immediately know if someone tries to cheat. No CTF or CLA is required. To see how this works, let's try to cheat.

If someone tries to stuff the ballot, Alice will detect the attempt in step (3) when she receives more votes than people. If Alice tries to stuff the ballot, Bob will notice in step (4).

More devious is to substitute one vote for another. Since the votes are encrypted with various public keys, anyone can create as many valid votes as needed. The decryption protocol has two rounds: round one consists of steps (3) through (7), and round two consists of steps (8) through (11). Vote substitution is detected differently in the different rounds.

If someone substitutes one vote for another in round two, his actions are discovered immediately. At every step the votes are signed and sent to all the voters. If one (or more) of the voters noticed that his vote is no longer in the set of votes, he immediately stops the protocol. Because the votes are signed at every step, and because everyone can backtrack through the second round of the protocol, it is easy to detect who substituted the votes.

identification number.

Step (6) gives each voter the capability to check that the CTF received his vote accurately. If his vote is miscounted, he can prove his case in step (9). Assuming a voter's vote is correct in step (6), the message he sends in step (9) constitutes a proof that his vote is miscounted.

One problem with the protocol is that a corrupt CTF could allocate the votes of people who respond in step (2) but who do not actually vote. Another problem is the complexity of the ANDOS protocol. The authors recommend dividing a large population of voters into smaller populations, such as election districts.

Another, more serious problem is that the CTF can neglect to count a vote. This problem cannot be resolved: Alice claims that the CTF intentionally neglected to count her vote, but the CTF claims that the voter never voted.

Voting without a Central Tabulating Facility

The following protocol does away with the CTF entirely; the voters watch each other. Designed by Michael Merritt [452, 1076, 453], it is so unwieldy that it cannot be implemented practically for more than a handful of people, but it is useful to learn from nevertheless.

Alice, Bob, Carol, and Dave are voting yes or no (0 or 1) on a particular issue. Assume each voter has a public and private key. Also assume that everyone knows everyone else's public keys.

(1) Each voter chooses his vote and does the following:
- **(a)** He attaches a random string to his vote.
- **(b)** He encrypts the result of step (a) with Dave's public key.
- **(c)** He encrypts the result of step (b) with Carol's public key.
- **(d)** He encrypts the result of step (c) with Bob's public key.
- **(e)** He encrypts the result of step (d) with Alice's public key.
- **(f)** He attaches a new random string to the result of step (e) and encrypts it with Dave's public key. He records the value of the random string.
- **(g)** He attaches a new random string to the result of step (f) and encrypts it with Carol's public key. He records the value of the random string.
- **(h)** He attaches a new random string to the result of step (g) and encrypts it with Bob's public key. He records the value of the random string.
- **(i)** He attaches a new random string to the result of step (h) and encrypts it with Alice's public key. He records the value of the random string.

If E is the encryption function, R_i is a random string, and V is the vote, his message looks like:

$$E_A(R_5,E_B(R_4,E_C(R_3,E_D(R_2,E_A(E_B(E_C(E_D(V,R_1)))))))$$

Each voter saves the intermediate results at each point in the calculation. These results will be used later in the protocol to confirm that his vote is among those being counted.

(2) Each voter sends his message to Alice.

(3) Alice decrypts all of the votes with her private key and then removes all of the random strings at that level.

(4) Alice scrambles the order of all the votes and sends the result to Bob.

Each vote now looks like this:

$$E_B(R_4,E_C(R_3,E_D(R_2,E_A(E_B(E_C(E_D(V,R_1)))))))$$

(5) Bob decrypts all of the votes with his private key, checks to see that his vote is among the set of votes, removes all the random strings at that level, scrambles all the votes, and then sends the result to Carol.

Since the anonymous key distribution protocol prevents the CTF from knowing which voter got which validation number, there is no way for the CTF to correlate validation numbers with votes received. The CTF still has to be trusted not to give validation numbers to ineligible voters, though. You can also solve this problem with blind signatures.

Improved Voting with a Single Central Facility

This protocol also uses ANDOS [1175]. It satisfies all six requirements of a good voting protocol. It doesn't satisfy the seventh requirement, but has two properties additional to the six listed at the beginning of the section:

> **7.** A voter can change his mind (i.e., retract his vote and vote again) within a given period of time.
> **8.** If a voter finds out that his vote is miscounted, he can identify and correct the problem without jeopardizing the secrecy of his ballot.

Here's the protocol:

> **(1)** The CTF publishes a list of all legitimate voters.
> **(2)** Within a specified deadline, each voter tells the CTF whether he intends to vote.
> **(3)** The CTF publishes a list of voters participating in the election.
> **(4)** Each voter receives an identification number, *I*, using an ANDOS protocol.
> **(5)** Each voter generates a public-key/private-key key pair: *k, d.* If *v* is the vote, he generates the following message and sends it to the CTF:
>
> $I,E_k(I,v)$
>
> This message must be sent anonymously.
> **(6)** The CTF acknowledges receipt of the vote by publishing:
>
> $E_k(I,v)$
>
> **(7)** Each voter sends the CTF:
>
> I,d
>
> **(8)** The CTF decrypts the votes. At the end of the election, it publishes the results of the election and, for each different vote, the list of all $E_k(I,v)$ values that contained that vote.
> **(9)** If a voter observes that his vote is not properly counted, he protests by sending the CTF:
>
> $I,E_k(I,v),d$
>
> **(10)** If a voter wants to change his vote (possible, in some elections) from *v* to *v'*, he sends the CTF:
>
> $I,E_k(I,v'),d$

A different voting protocol uses blind signatures instead of ANDOS, but is essentially the same [585]. Steps (1) through (3) are preliminary to the actual voting. Their purpose is to find out and publicize the total number of actual voters. Although some of them probably will not participate, it reduces the ability of the CTF to add fraudulent votes.

In step (4), it is possible for two voters to get the same identification number. This possibility can be minimized by having far more possible identification numbers than actual voters. If two voters submit votes with the same identification tag, the CTF generates a new identification number, *I'*, chooses one of the two votes, and publishes:

$I',E_k(I,v)$

The owner of that vote recognizes it and sends in a second vote, by repeating step (5), with the new

One solution is to divide the CTF in two. Neither party would have the power to cheat on its own.

The following protocol uses a Central Legitimization Agency (CLA) to certify voters and a separate CTF to count votes [1373].

(1) Each voter sends a message to the CLA asking for a validation number.
(2) The CLA sends the voter back a random validation number. The CLA maintains a list of validation numbers. The CLA also keeps a list of the validation numbers' recipients, in case someone tries to vote twice.
(3) The CLA sends the list of validation numbers to the CTF.
(4) Each voter chooses a random identification number. He creates a message with that number, the validation number he received from the CLA, and his vote. He sends this message to the CTF.
(5) The CTF checks the validation number against the list it received from the CLA in step (3). If the validation number is there, the CTF crosses it off (to prevent someone from voting twice). The CTF adds the identification number to the list of people who voted for a particular candidate and adds one to the tally.
(6) After all votes have been received, the CTF publishes the outcome, as well as the lists of identification numbers and for whom their owners voted.

Like the previous protocol, each voter can look at the lists of identification numbers and find his own. This gives him proof that his vote was counted. Of course, all messages passing among the parties in the protocol should be encrypted and signed to prevent someone from impersonating someone else or intercepting transmissions.

The CTF cannot modify votes because each voter will look for his identification string. If a voter doesn't find his identification string, or finds his identification string in a tally other than the one he voted for, he will immediately know there was foul play. The CTF cannot stuff the ballot box because it is being watched by the CLA. The CLA knows how many voters have been certified and their validation numbers, and will detect any modifications.

Mallory, who is not an eligible voter, can try to cheat by guessing a valid validation number. This threat can be minimized by making the number of possible validation numbers much larger than the number of actual validation numbers: 100-digit numbers for a million voters, for example. Of course, the validation numbers must be generated randomly.

Despite this, the CLA is still a trusted authority in some respects. It can certify ineligible voters. It can certify eligible voters multiple times. This risk could be minimized by having the CLA publish a list of certified voters (but not their validation numbers). If the number of voters on this list is less than the number of votes tabulated, then something is awry. However, if more voters were certified than votes tabulated, it probably means that some certified people didn't bother voting. Many people who are registered to vote don't bother to cast ballots.

This protocol is vulnerable to collusion between the CLA and the CTF. If the two of them got together, they could correlate databases and figure out who voted for whom.

Voting with a Single Central Facility

A more complex protocol can be used to overcome the danger of collusion between the CLA and the CTF [1373]. This protocol is identical to the previous one, with two modifications:

— The CLA and the CTF are one organization, and
— ANDOS (see Section 4.13) is used to anonymously distribute validation numbers in step (2).

completely. It's analogous to having an election judge staring over your shoulder in the voting booth.

These two examples show how difficult it is to achieve the first three requirements of a secure voting protocol, let alone the others.

Voting with Blind Signatures

We need to somehow dissociate the vote from the voter, while still maintaining authentication. The blind signature protocol does just that.

> **(1)** Each voter generates 10 sets of messages, each set containing a valid vote for each possible outcome (e.g., if the vote is a yes or no question, each set contains two votes, one for "yes" and the other for "no"). Each message also contains a randomly generated identification number, large enough to avoid duplicates with other voters.
> **(2)** Each voter individually blinds all of the messages (see Section 5.3) and sends them, with their blinding factors, to the CTF.
> **(3)** The CTF checks its database to make sure the voter has not submitted his blinded votes for signature previously. It opens nine of the sets to check that they are properly formed. Then it individually signs each message in the set. It sends them back to the voter, storing the name of the voter in its database.
> **(4)** The voter unblinds the messages and is left with a set of votes signed by the CTF. (These votes are signed but unencrypted, so the voter can easily see which vote is "yes" and which is "no.")
> **(5)** The voter chooses one of the votes (ah, democracy) and encrypts it with the CTF's public key.
> **(6)** The voter sends his vote in.
> **(7)** The CTF decrypts the votes, checks the signatures, checks its database for a duplicate identification number, saves the serial number, and tabulates the votes. It publishes the results of the election, along with every serial number and its associated vote.

A malicious voter, call him Mallory, cannot cheat this system. The blind signature protocol ensures that his votes are unique. If he tries to send in the same vote twice, the CTF will notice the duplicate serial number in step (7) and throw out the second vote. If he tries to get multiple votes signed in step (2), the CTF will discover this in step (3). Mallory cannot generate his own votes because he doesn't know the facility's private key. He can't intercept and change other people's votes for the same reason.

The cut-and-choose protocol in step (3) is to ensure that the votes are unique. Without that step, Mallory could create a set of votes that are the same except for the identification number, and have them all validated.

A malicious CTF cannot figure out how individuals voted. Because the blind signature protocol prevents the facility from seeing the serial numbers on the votes before they are cast, the CTF cannot link the blinded vote it signed with the vote eventually cast. Publishing a list of serial numbers and their associated votes allows voters to confirm that their vote was tabulated correctly.

There are still problems. If step (6) is not anonymous and the CTF can record who sent in which vote, then it can figure out who voted for whom. However, if it receives votes in a locked ballot box and then tabulates them later, it cannot. Also, while the CTF may not be able to link votes to individuals, it can generate a large number of signed, valid votes and cheat by submitting those itself. And if Alice discovers that the CTF changed her vote, she has no way to prove it. A similar protocol, which tries to correct these problems, is [1195, 1370].

Voting with Two Central Facilities

Chapter 6
Esoteric Protocols

6.1 Secure Elections

Computerized voting will never be used for general elections unless there is a protocol that both maintains individual privacy and prevents cheating. The ideal protocol has, at the very least, these six requirements:

1. Only authorized voters can vote.
2. No one can vote more than once.
3. No one can determine for whom anyone else voted.
4. No one can duplicate anyone else's vote. (This turns out to be the hardest requirement.)
5. No one can change anyone else's vote without being discovered.
6. Every voter can make sure that his vote has been taken into account in the final tabulation.

Additionally, some voting schemes may have the following requirement:

7. Everyone knows who voted and who didn't.

Before describing the complicated voting protocols with these characteristics, let's look at some simpler protocols.

Simplistic Voting Protocol #1

(1) Each voter encrypts his vote with the public key of a Central Tabulating Facility (CTF).
(2) Each voter sends his vote in to the CTF.
(3) The CTF decrypts the votes, tabulates them, and makes the results public.

This protocol is rife with problems. The CTF has no idea where the votes are from, so it doesn't even know if the votes are coming from eligible voters. It has no idea if eligible voters are voting more than once. On the plus side, no one can change anyone else's vote; but no one would bother trying to modify someone else's vote when it is far easier to vote repeatedly for the result of your choice.

Simplistic Voting Protocol #2

(1) Each voter signs his vote with his private key.
(2) Each voter encrypts his signed vote with the CTF's public key.
(3) Each voter sends his vote to a CTF.
(4) The CTF decrypts the votes, checks the signatures, tabulates the votes, and makes the results public.

This protocol satisfies properties one and two: Only authorized voters can vote and no one can vote more than once—the CTF would record votes received in step (3). Each vote is signed with the voter's private key, so the CTF knows who voted, who didn't, and how often each voter voted. If a vote comes in that isn't signed by an eligible voter, or if a second vote comes in signed by a voter who has already voted, the facility ignores it. No one can change anyone else's vote either, even if they intercept it in step (3), because of the digital signature.

The problem with this protocol is that the signature is attached to the vote; the CTF knows who voted for whom. Encrypting the votes with the CTF's public key prevents anyone from eavesdropping on the protocol and figuring out who voted for whom, but you have to trust the CTF