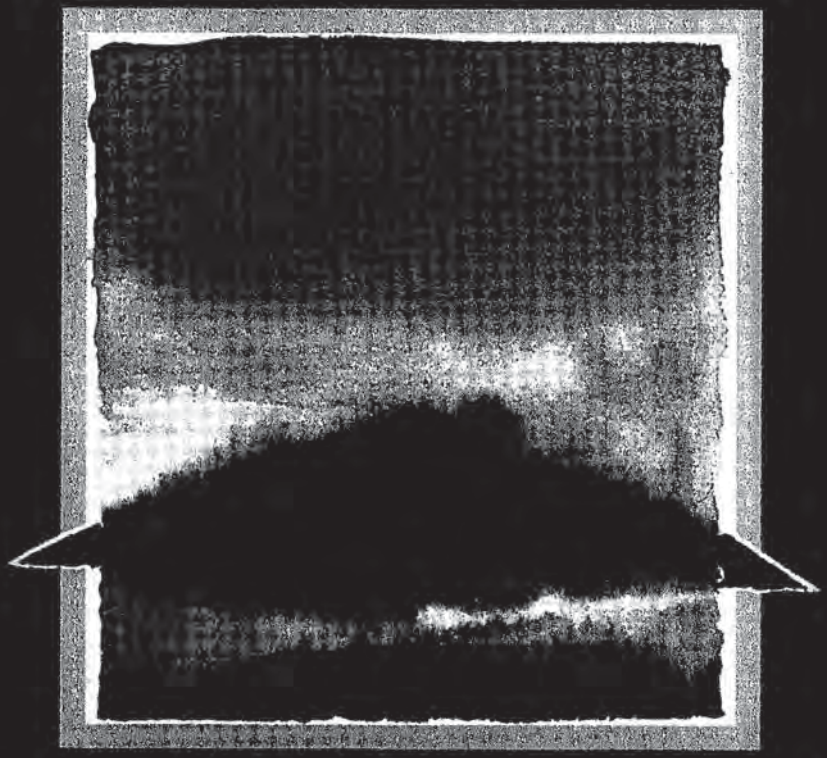
APPLIED CRYPTOGRAPHY



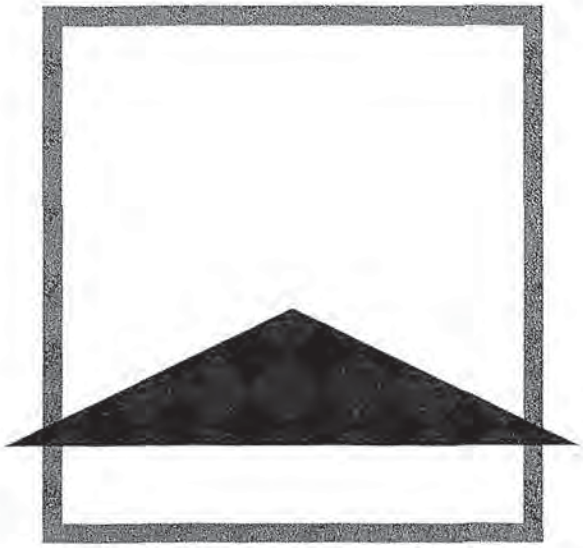
Protocols, Algorithms, and Source Code in C

BRUCE SCHNEIER

ZANOWSICY

Applied Cryptography

Protocols, Algorithms, and Source Code in C



Applied Cryptography

Protocols, Algorithms, and Source Code in C

Bruce Schneier

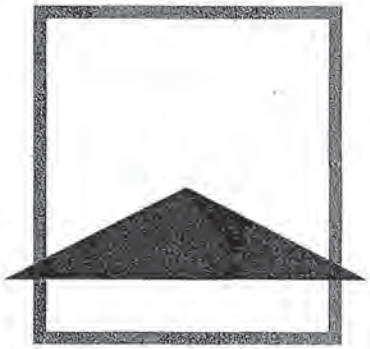
John Wiley & Sons, Inc.
NEW YORK • CHICHESTER • BRISBANE • TORONTO • SINGAPORE

Associate Publisher: Katherine Schowalter
Editor: Paul Farrell
Managing Editor: Beth Austin
Editorial Production & Design: Editorial Services of New England, Inc.

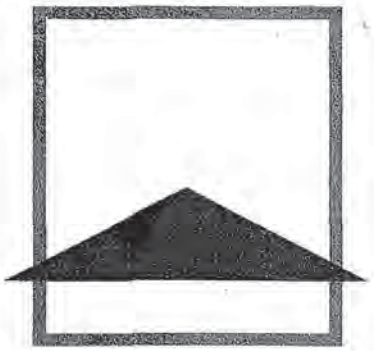
Library of Congress Cataloging-in-Publication Data
Schneier, Bruce
Applied cryptography : protocols, algorithms, and source code in C
/ Bruce Schneier.
p. cm.
Includes bibliographical references and index.
ISBN 0-471-59756-2 (paper)
1. Computer security. 2. Telecommunication—security measures. 3. Cryptography. 4. Title

QA76.9.A25S35 1993
005.8'2—dc20 93-2139
CIP

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1



Contents



Foreword

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 20th 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 [355]. 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 [418a], the device destined to be a mainstay of military cryptography for nearly fifty 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 [804]. 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* [462]. 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, and by the early 1970s several technical reports on this subject by Feistel and his colleagues had been made public by IBM [339,837,839].

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 integration 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 much 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 [299], one of the indirect aspects of our contribution was to introduce a problem that does not even appear easy to solve. Now an aspiring cryptosystem designer could produce something that would be recognized as clever — something that did more than just turn meaningful text into nonsense. The result has been a spectacular increase in the number of people working in cryptography, the number of meetings held, and the number of books and papers published.

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

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

This is the gap that Bruce Schneier's *Applied Cryptography: Protocols, Algorithms, and Source Code in C* 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 fifteen years of public research. The table of contents tells 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 National Security Agency.

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 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.

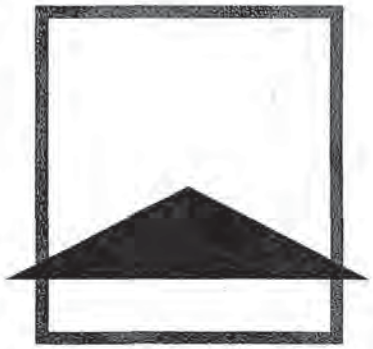
As the eighties progressed, pressure focused more on the practice than the study of cryptography. Existing laws gave the NSA the power, through the Department of State, to regulate the export of cryptographic equipment. As business became more and more international and the American fraction of the world 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 of what was sold in the United States.

As this is written, a new challenge confronts the public practice of cryptography. The government proposes to replace 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 proposal range from a potentially disastrous impact on personal privacy to the high cost of having to add hardware to products that had previously encrypted in software. It has attracted widespread negative comment, especially from 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 recent 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 gray area" with respect to the law. If that is so, it is a gray area on which this book can be expected to shed some light.

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 might 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 might.

Whitfield Diffie
Mountain View, CA



Preface

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

If I take a letter, lock it in a safe, and hide the safe somewhere in New York . . . and 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 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, and its counterparts in the 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. The private individual, with far less expertise and budget, has been powerless to protect his own privacy against these governments.

During the last twenty years, there has been an explosion of public academic research in cryptography. For the first time, state-of-the-art computer cryptography is being practiced outside the secured walls of the military agencies. It is now possible for you and me to employ security practices that can protect against the most powerful of adversaries—security that may even protect against the military agencies.

Does the average person really need this kind of security? I say yes. He may be planning a political campaign, discussing his taxes, or having an illicit affair. He may be designing a new product, discussing a marketing strategy, or planning a hostile business takeover. He may be living in a country that does not respect the rights of privacy of its citizens. He may be doing something that he feels

shouldn't be illegal, but is. Whatever his reasons, his data and communications are personal, private, and nobody's business but his own.

This book is being published in a tumultuous time. The Clinton Administration has just proposed the Clipper chip, which ensures the government the ability to conduct electronic surveillance. Clipper is based on the Orwellian assumption that the government has the right to listen to private communications. It promotes the power of the government over the power of the individual. This is not simply a little government proposal in some obscure area; it is a preemptive and unilateral attempt to usurp powers that previously belonged to the people.

Clipper does not protect privacy; it forces individuals to unconditionally trust that the government will respect their privacy. It assumes that the government is the good guy and private citizens are all bad guys. The same law enforcement authorities that illegally tapped Martin Luther King, Jr.'s phones can easily tap a phone protected with Clipper. During the past five years, local police authorities have been either charged criminally or sued civilly in numerous jurisdictions — including Maryland, Connecticut, Vermont, Georgia, Missouri, and Nevada —for conducting illegal wiretaps. It's a poor idea to deploy technologies that could someday facilitate a police state.

At the time I am writing, it is too early to tell what will happen to the Clipper chip. At one extreme, it could become a minor anecdote about a failed attempt by the government to invade peoples' privacy—people reading this five years from now will wonder what I am talking about. At the other, it could be the first in a series of initiatives designed to outlaw private encryption—five years from now most of the techniques described in this book will be illegal. The truth will probably lie somewhere in the middle.

Encryption is too important to be left to the government.

HOW TO READ THIS BOOK

I wrote *Applied Cryptography* to be a comprehensive reference work for modern cryptography. The first chapter introduces cryptography, defines many terms, and briefly discusses pre-computer cryptography.

Part 1 (Chapters 2 through 6) discuss cryptographic protocols. This is what you 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). Some of these protocols are obvious; others are almost amazing. If you are interested, read through all of the things cryptography can do that seem impossible. If these protocols start getting tedious, skip to the next section.

Part II discusses cryptographic techniques. Both chapters in this section are important for even the most basic uses of cryptography. Chapter 7 is about keys: how long should a key be in order to be secure, how do you generate keys, how do you store keys, how do you dispose of keys, etc. Key management is the hardest part of cryptography, and often the Achilles heel of an otherwise secure system. Chapter 8 discusses different ways of using cryptographic algorithms, and how to choose algorithms for communication, data storage, etc.

In Part III, the book finally turns to algorithms. Chapter 9 is a 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 this chapter. Chapter 10 discusses DES. Chapter 11 discusses other symmetric algorithms. If you want something more secure than DES, skip to the section on IDEA. If you want to read about a bunch of algorithms, some of which may be more secure than DES, read the whole chapter. Chapters 12 and 13 discuss public-key algorithms; if you just want the basics, read the introduction to Chapter 12, the section on Diffie-Hellman, the section on RSA, and the section in Chapter 13 on DSS. Chapter 14 is about one-way hash functions: MD5 and SHA are the most common, although I discuss many more. Chapter 15 is about random number generation and stream ciphers. There is a bias in cryptographic research in the world; most of the work on stream ciphers is done in Europe, while in the United States stream cipher research takes a back seat to block cipher research. To some degree, this book perpetuates the bias. And finally, Chapter 16 discusses algorithms specifically designed to implement some of the protocols in Part I. The math can get complicated here; wear your seat belt.

Part IV has two chapters. Chapter 17 discusses some real-world implementations of these algorithms and protocols, and chapter 18 touches on some of the political issues surrounding cryptography. These are by no means intended to be comprehensive.

Finally, Part V gives source code listings for some of the algorithms in Part IV. I wanted to include much more code than I was able to, but space limitations got in the way. There is an associated source code disk that includes these listings, as well as all the listings that I didn't have space for. If you are at all interested in implementing or playing with the cryptographic algorithms in this book, get the disk.

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 who are interested in formalism, there are copious pointers to references in the academic literature.

If there was a criticism of this book, it 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. There is a lot being done in the field; this is the first time so much of it has been gathered under one roof. For those who are more interested in a tutorial, I apologize. Even so, there are many things that space forced me to leave out. I tried to cover topics that I felt were important, topics that I felt were practical, and topics that I felt were interesting. If I couldn't cover a topic in depth, I gave references to articles and papers that did. If there are aspects of cryptography that you wish were covered in more depth, I apologize.

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. Any errors I find after publication will be corrected on the source code disk. Any new information discovered after publication will also be on the disk. If someone finds an error, please tell me. The first person to find each error in the book will get a free copy of the source code disk.

ACKNOWLEDGMENTS

The list of people who had a hand in this book seems unending, but all are worthy of mention. I would like to thank Don Alvarez, Ross Anderson, Karl Barrus, Steve Bellovin, Dan Bernstein, Eli Biham, Joan Boyar, Karen Cooper, Whitfield Diffie, Joan Feigenbaum, Phil Karn, Neal Koblitz, Xuija Lai, Tom Leranth, Mark Markowitz, Ralph Merkle, Bill Patton, Peter Pearson, Mark Riordan, and Marc Schwartz for reading and editing all or parts of the manuscript; Lawrie Brown, Leisa Condie, Peter Gutmann, Alan Insley, Xuija Lai, Peter Pearson, Ken Pizzini, Richard Outerbridge, RSA Data Security Inc., Michael Wood, and Phil Zimmermann for providing source code; the readers of sci.crypt for commenting on ideas and answering questions, Paul MacNerland for creating the figures; Randy Seuss for providing Internet access; Jeff Duntemann and Jon Erickson for helping me find a publisher; Paul Farrell for editing this book; 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. These people helped to create a far better book than I could have done alone.

Bruce Schneier
Oak Park, Ill.

Foundations

1.1 TERMINOLOGY

Sender and Receiver

Suppose someone, whom we shall call the **sender,** wants to send a **message** to someone else, whom we shall call the **receiver.** Moreover, the sender wants to make sure an intermediary cannot affect the message in any way: specifically, the receiver cannot intercept and read the message, intercept and modify the message, or fabricate a realistic-looking substitute message.

Messages and Encryption

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

The art and science of keeping messages secure is called **cryptography,** and it is practiced by **cryptographers. Cryptanalysts** are practitioners of **cryptanalysis,** the art and science of breaking ciphertext, i.e., seeing through the disguise. The branch of mathematics embodying both cryptography and cryptanalysis is called **cryptology,** and its practitioners are called **cryptologists.** These days almost all cryptologists are also theoretical mathematicians—they have to be.

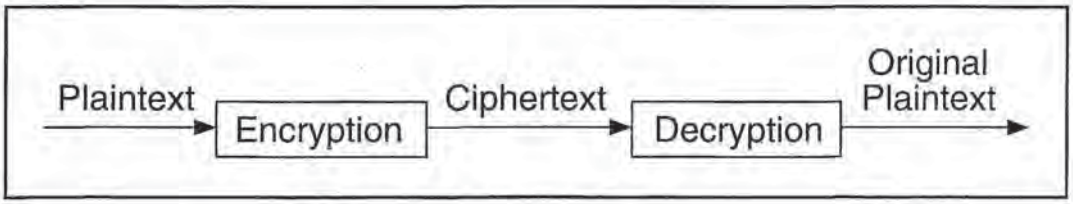


Figure 1.1
Encryption and decryption.

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

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

$$E(P) = C$$

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

$$D(C) = P$$

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

$$D(E(P)) = P$$

Algorithms and Ciphers

A **cryptographic algorithm,** also called a **cipher,** is the mathematical function used for encryption and decryption. To encrypt a plaintext message, apply an **encryption algorithm** to the plaintext. To decrypt a ciphertext message, apply a **decryption algorithm** to the ciphertext.

If the security of an algorithm is based on keeping the nature of the algorithm secret, it is called **restricted.** Restricted algorithms have historical interest, but by today's data security standards they provide woefully inadequate security. A large or changing group of users cannot use them, because users will eventually reveal the secret. When they do, the whole security of the system fails. More important, most restricted cryptosystems are trivial to break by experienced cryptanalysts. Despite this, restricted algorithms are enormously popular for low-security applications. Zenith's video-scrambling technique is an example of a restricted algorithm.

For real security, all modern encryption algorithms use a **key,** denoted by k. This key can take on one of many values (a large number is best). The range of possible values of the key is called the **keyspace.**

The value of the key affects the encryption and decryption functions, so the encryption and decryption functions now become:

$$E_k(P) = C$$
$$D_k(C) = P$$

And if the encryption key and the decryption key are the same, then:

$$D_k(E_k(P)) = P$$

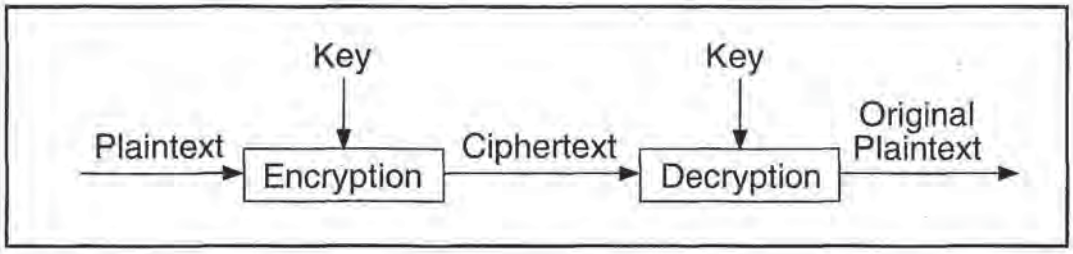


Figure 1.2
Encryption and decryption with key.

This is shown in Figure 1.2.

There are algorithms where the encryption key and the decryption key come in pairs (see Figure 1.3). That is, the encryption key, k_1, is different from the corresponding decryption key, k_2. In this case:

$$E_{k_1}(P) = C$$
$$D_{k_2}(C) = P$$
$$D_{k_2}(E_{k_1}(P)) = P$$

Symmetric Algorithms and Public-Key Algorithms

There are two general forms of key-based algorithms: symmetric and public-key. **Symmetric algorithms** are algorithms where the encryption key can be calculated from the decryption key and vice versa. In many such systems, 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 the sender and receiver to agree on a key before they pass messages back and forth. This key must be kept secret. The security of a symmetric algorithm rests in the key; divulging the key means that anybody could encrypt and decrypt messages in this cryptosystem.

Encryption and decryption with a symmetric algorithm are denoted by:

$$E_k(P) = C$$
$$D_k(C) = P$$

Symmetric algorithms can be divided into two categories. Some operate on the plaintext a single bit 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 algorithms that are implemented on computer, a typical block size is 64 bits—large enough to preclude analysis and small enough to be workable. In both block and stream algorithms, the same key is used for both encryption and

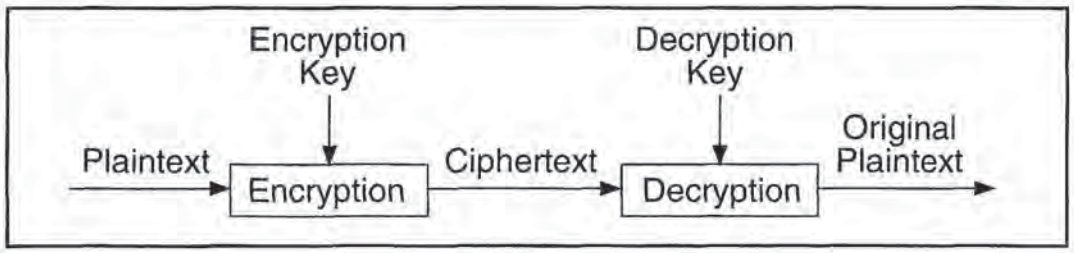


Figure 1.3
Encryption and decryption with two keys.

decryption. (Before computers, algorithms generally operated on plaintext one character at a time. You can either think of this as a stream algorithm operating on a stream of characters or as a block algorithm operating on 8-bit blocks.)

Public-key algorithms are different. They are designed so that the key used for encryption is different from the key used for decryption. Furthermore, the decryption key cannot be (at least in any reasonable amount of time) calculated from the encryption key. They are called public-key systems because the encryption key can be made public: a complete stranger can use the encryption key to encrypt a message, but only someone 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.** In instances when both keys must be kept secret, sometimes the terms "encryption key" and "decryption key" will be used. The private key is sometimes also called the **secret key,** but to avoid confusion with symmetric algorithms, that moniker won't be used here.

Encryption using public key k is denoted by:

$$E_k(P) = C$$

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

$$D_k(C) = P$$

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 will be denoted by, respectively:

$$E_k(P) = C$$
$$D_k(C) = P$$

In this book, "algorithm" will refer specifically to the mathematical transformations for encryption and decryption. "Cryptosystem" will refer to the algorithm, plus the way in which it is implemented. "Cipher" will often be used to refer to a family of algorithms, e.g., "block cipher."

Cryptanalysis

The primary purpose 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**). Cryptanalysis is the science of recovering the plaintext of a message without the key. Successful cryptanalysis may recover the plaintext or the key. It also may find weaknesses in a cryptosystem that eventually lead to the above results.

An attempted cryptanalysis is called an **attack.** A successful attack is called a **method.** An attack assumes that the cryptanalyst has the details of the cryptographic algorithm. While this is not always the case in real-life

cryptanalysis, it is the conventional assumption for academic cryptanalysis. It is a good assumption to make; if your security depends on the secrecy of the algorithm, then there is only minimal security.

There are six general types of cryptanalytic attacks, listed in order of power. Each of them assumes that the cryptanalyst has complete knowledge of the encryption algorithm used:

1. **Ciphertext-only attack.** In this 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 deduce the key (or keys) used to encrypt the messages in order to decrypt other messages encrypted with the same keys.

   Given: $C_1 = E_k(P_1), C_2 = E_k(P_2), \ldots C_i = E_k(P_i)$
   Deduce: Either $P_1, P_2, \ldots P_i$; k; or an algorithm to infer P_{i+1} from $C_{i+1} = E_K(P_{i+1})$

2. **Known-plaintext attack.** The cryptanalyst not only has access to the ciphertext of several messages but also to the plaintext of those messages. The 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.

   Given: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2), \ldots P_i, C_i = E_k(P_i)$
   Deduce: Either k, or an algorithm to infer P_{i+1} from $C_{i+1} = E_k(P_{i+1})$

3. **Chosen-plaintext attack.** Cryptanalysts not only have access to the ciphertext and associated plaintext for several messages, but they also choose the encrypted plaintext. This is more powerful than a known-plaintext attack, because cryptanalysts can choose specific plaintext blocks to encrypt, ones that might yield more information about the key. The 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.

   Given: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2), \ldots P_i, C_i = E_k(P_i)$, where the cryptanalysts choose $P_1, P_2, \ldots P_i$
   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 cryptanalysts choose the plaintext that is encrypted, but they can modify the choice based on the results of previous encryption. In a chosen-plaintext attack, cryptanalysts might just be able to choose one large block of plaintext to be encrypted; in an adaptive-chosen-plaintext attack they can choose a smaller block of plaintext and then choose another based on the results of the first, etc.

5. **Chosen-ciphertext attack.** Cryptanalysts can choose different ciphertexts to be decrypted and have access to the decrypted plaintext. In an instance

when cryptanalysts have a tamperproof box that does automatic decryption, the job is to deduce the key.

Given: $C_1, P_1 = D_k(C_1), C_2, P_2 = D_k(C_2), \ldots C_i, P_i = D_k(C_i)$
Deduce: k

This attack is primarily applicable to public-key cryptosystems and will be discussed in Section 12.4. A chosen-ciphertext attack also works against symmetric algorithm, but due to the symmetry of these cryptosystems it is equivalent in complexity to a chosen-plaintext attack.

6. **Chosen-key attack.** This is not an attack when you're given the key. It's strange and obscure, not very practical, and is discussed in Section 10.1.

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 the United States 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, called protocols, loop structures, etc. David Kahn's books [462,463,464] have some historical examples of these kinds of attacks.

One of the fundamental axioms of cryptography is that the enemy is in full possession of the details of the algorithm and lacks only the specific key used in the encryption. (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 this is not always true in real-world cryptanalysis, it is always true in academic cryptanalysis; and it's a good assumption to make in real-world cryptanalysis. 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.

Cryptanalysts don't always have access to the algorithm—for example, when the United States broke the Japanese diplomatic code, PURPLE, during World War II [462]—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. There have been many historical instances when cryptanalysts did not know the encryption algorithms; sometimes they broke the algorithm anyway, and sometimes they did not. In any case, it is unrealistic to rely on it.

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.

On the other hand, a good algorithm can be made public without worry. You can send it to your adversaries, publish it in a magazine, or shout it from the rooftops. It doesn't matter; even the designer of the algorithm can't decrypt messages without the key.

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

Security of Cryptosystems

Different cryptosystems have different levels of security, depending on how hard they are to break. As we will see, all algorithms but one are theoretically **breakable,** given enough time and computing resources. If the time and money required to break an algorithm is more than the value of the encrypted data, then it is probably safe. Computers are becoming increasingly faster and cheaper. At the same time, the value of the data decreases over time. It is important that those two lines never cross.

Some algorithms are only breakable with the benefit of more time than the universe has been in existence and a computer larger than all the matter in the universe. These algorithms are theoretically breakable, but not breakable in practice. An algorithm that is not breakable in practice is **secure.**

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

The amount of computing time and power required to recover the encryption key is called the **work factor,** and is expressed as an order of magnitude. If an algorithm has a work factor of 2^{128}, then 2^{128} operations are required to break the algorithm. These operations can be very complex and time-consuming, but details of the operations are left to the implementation. Still, if you assume that you have enough computing speed to perform a million of them every second, and you set a million parallel processors against the task, it will still take over 10^{19} years to recover the key. (For comparison's sake, the age of the universe is estimated at 10^{10} years.) I would consider an algorithm that takes a billion times the age of the universe to break to be computationally secure.

While the work factor required to break a given algorithm is constant (that is, until some cryptanalyst finds a better cryptanalytic attack), computing power is anything but. During the last half-century we have seen phenomenal advances in computing power, and there is no reason to think that it will change anytime soon. Many cryptanalytic attacks are perfect for parallel machines: the task can be broken down into billions of tiny pieces and none of the processors needs to interact with another. Pronouncing that an algorithm is secure simply because it is unfeasible to break, given current technology, is dicey at best. Good cryptosystems are designed to be unfeasible to break with the computing power that is expected to evolve many years in the future.

Historical Terms

There are other cryptographic terms. A cryptosystem is also called a **code** or a **cipher.** Encrypting is also called **encoding** or **enciphering,** and decrypting is also called **decoding** or **deciphering.**

Historically, q code refers to a cryptosystem that deals with linguistic units: words, phrases, sentences, etc. For example, the word "MULBERRY" 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; they are discussed in [462,463].

The word "cipher" has historically been used to refer to cryptosystems in which individual letters are swapped and substituted or otherwise scrambled to hide the plaintext. This is what this book is about.

Ciphers were useful because they were general-purpose. If there was no entry in a codebook for "ANTEATERS," then you couldn't say it. On the other hand, any message can be encrypted with the cipher.

1.2 CLASSICAL CRYPTOGRAPHY

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

Things are more complex these days, but the philosophy has remained 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 and nothing more. Most good cryptographic algorithms still combine elements of substitution and transposition (there are exceptions).

1.2.1 Substitution Ciphers and Transposition Ciphers

Substitution Ciphers

A **substitution cipher** is one in which each character in the plaintext is substituted for another character in the ciphertext. This substitution serves to obscure the plaintext from everyone but the recipient, who inverts the substitution on the ciphertext to recover the plaintext.

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

- A **simple substitution cipher** is one in which a 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, etc.

- 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.
- 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," etc.

The famous **Caesar Cipher,** in which each plaintext character is replaced by the character three to the right mod 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.

ROT13 is a simple encryption program commonly found on UNIX systems. In this cipher, A is replaced by N, B is replaced by O, etc. Every letter is rotated thirteen places. This is a simple substitution cipher.

```
#include  <stdio.h>
main()   /* streamlined version of copy input to output */
{
  int c;
  while ((c =getchar()) !=EOF)
     {
      if( c>= 'a' && c<= 'm'  )
        c=c+13;
    else if( c >= 'n'  && c <= 'z'  )
      c=c-13;
    else if( c >= 'A'  && c <= 'M'  )
      c=c+13;
    else if( c >= 'N'  && c <= 'Z'  )
      c=c-13;
    putchar(c);
  }
}
```

Encrypting a file twice with ROT13 restores the original file.

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

ROT13 is not intended for security; it is often used in electronic mail messages to hide potentially offensive text, to avoid giving away the solution to a puzzle, etc.

These 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 [806]. A general algorithm for solving these sorts of ciphers can be found in [689].

Homophonic substitution ciphers were used as early as 1401 by the Duchy of Mantua [462]. 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 attack is harder, but only takes a few seconds on a computer. Details are in [710].

Polyalphabetic substitution ciphers were invented by Leon Battista in 1568 [462]. They were used by the Union army during the American Civil War. Despite the fact that they can be broken easily [475,355,360,462] (especially with the help of computers), many commercial computer security products use ciphers of this form [744]. (Details on how to break this encryption scheme as it was found in the WordPerfect word-processing program versions can be found in [80,84].) The Vigenère cipher and the Beaufort cipher are 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. With computers, there are techniques that can easily break substitution ciphers with very long periods.

A **running-key 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 [354,462].

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 [462]. It encrypts pairs of letters together. Its cryptanalysis is discussed in [360,832,500]. The Hill cipher is another example of a polygram substitution cipher [436].

Transposition Ciphers

A **transposition** cipher is one in which the characters in the plaintext remain the same, but their order 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 [360,832].

The German ADFGVX cipher, used during World War I, is a transposition cipher (plus a simple substitution). It was a very complex algorithm for its day but was broken by Georges Painvin, a French cryptanalyst [462].

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

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

```
COMPUTERGR
APHICSMAYB
ESLOWBUTAT
LEASTITSEX
PENSIVE
```

Ciphertext: CAELP OPSEE MHLAN PIOSS UCWTI TSBIU EMUTE RATSG YAERB TX

Figure 1.4
Columnar transposition cipher.

Rotor Machines

In the 1920s, various mechanical encryption devices were invented to automate the process of encryption. They were based on the concept of a **rotor,** a mechanical wheel that was wired to perform a general cryptographic substitution. For example, a rotor might be wired to substitute "F" for "A," "U" for "B," "L" for "C," etc. These devices used multiple rotors to implement a version of the Vigenère cipher with a very long period and were called **rotor machines.**

A rotor machine has a keyboard and a series of rotors. Each rotor has 26 positions and performs a simple substitution. The rotations of the rotors are a Caesar Cipher. It is the combination of all these rotors that makes the machine secure. Because the rotors all move, and at different rates, the period for an n-rotor machine is 26^n.

The best-known rotor device is the Enigma. The Enigma was used by the Germans during World War II. The basic idea was invented by Arthur Scherbius and Arvid Gerhard Damm in Europe. It was patented in the United States by Arthur Scherbius [772]. The Germans beefed up the basic design considerably for wartime use.

There was a plugboard that slightly permuted the plaintext. There was a reflecting rotor that forced each rotor to operate on each plaintext letter twice. As complicated as the Enigma was, it was broken during World War II. A team of Polish cryptographers broke a simplified Enigma; a British team, including Alan Turing, broke the actual Enigma. For explanations of how rotor ciphers work and how they were broken, see [462,45,301,258,500,740,873]. Two fascinating accounts of how the Enigma was broken are [438,464].

Further Reading

This is not a book about classical cryptography, so I will not dwell further on these subjects. Two excellent precomputer cryptology books are [360,832]. Dorothy Denning discusses many of these ciphers in [268], and [500] has some fairly complex mathematical analyses of the same ciphers. An article that presents a good overview of the subject is [356]. David Kahn's historical cryptography books are also excellent [462,463,464].

1.2.2 Computer Algorithms

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

- **DES** (Data Encryption Standard) is currently the most popular computer encryption algorithm. DES is a U.S.-government standard encryption algorithm and has been endorsed by the U.S. military for encrypting "Unclassified but Sensitive" information. 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.

1.2.3 Simple XOR

It's an embarrassment to put this algorithm in a book like this because it's nothing more than a Vigenère cipher. It is included because it is so prevalent in commercial software packages, at least those in the MS-DOS and Macintosh worlds [847]. Unfortunately, if a software security program proclaims that it has a "proprietary" encryption algorithm—one that is 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;
  int *cp;
  int c;

  if (cp = argv[1])  {
    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 same key is used for both encryption and decryption. The plaintext is being XORed with a keyword to generate the ciphertext. Since XORing the same value twice restores the original, encryption and decryption use exactly the same program:

$$P \text{ XOR } K = C$$
$$C \text{ XOR } K = P$$
$$(P \text{ XOR } K) \text{ XOR } K = P$$

There's no real security here. This kind of encryption is trivial to break, even without computers [360,832]. It will only take a few minutes with a computer.

Assume the plaintext is English. Furthermore, assume the key length is an arbitrary small number of bytes (although in the source code example, it is always eight bytes). Here's how to break it:

1. Discover the length of the key by a procedure known as **counting coincidences** [355]. Trying each byte displacement of the key against itself, count those bytes that are equal. If the two ciphertext portions have used the same key, something over 6% of the bytes will be equal. If they have used a different key, then less than 0.4% will be equal (assuming a random key encrypting normal ASCII text; other plaintext will have different numbers). The smallest displacement that indicates an equal key length is the length of the repeated key.
2. Shift the key by that length and XOR it with itself. This removes the key and leaves you with text XORed with itself. Since English has about one bit of real information per byte (see Section 9.1), there is plenty of redundancy for choosing a unique decryption.

Despite this, the list of software vendors that tout this sort of algorithm as being "almost as secure as DES" is staggering [774]. It might keep your kid sister from reading your files, but it won't stop a cryptographer for more than a few minutes.

1.2.4 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 [462]. In its classical form, 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. The sender uses each key letter on the pad to encrypt exactly one plaintext character. The receiver has an identical pad and uses each key on the pad, in turn, to decrypt each letter of the ciphertext.

Each key is used exactly once, for only one message. The sender encrypts the message and then destroys the pad's pages. The receiver does the same thing after decrypting the message. New message—new page and new key letters.

Assuming an adversary can't get access to the pages of the one-time pad used to encrypt the message, this scheme is perfectly secure. A given ciphertext message is equally likely to be any possible plaintext message of equal size. For example, if the message is:

ONETIMEPAD

and the key sequence from the pad is

TBFRGFARFM

then the ciphertext is

IPKLPSFHGQ

Since every key sequence is equally likely (remember, the keys are generated in a random manner), 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 XORed with 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 sequence. If you use a cryptographically weak algorithm to

generate your key sequence, there might be trouble. If you use a real random source—this is much harder than it might first appear—it is safe.

Using a pseudo-random number generator doesn't count; often they have nonrandom properties. Many of the stream ciphers described later in this book try to approximate this system with a pseudo-random sequence, but most fail. The sequences they generate only seem random, but careful analysis yields nonrandomness, which a cryptanalyst can exploit. However, it is possible to generate real random numbers, even with a microcomputer. Some techniques for how to do this will be covered in Section 15.3.

The idea of a one-time pad can easily be extended to the encryption of binary data. Instead of a one-time pad consisting of letters, use a one-time pad of bits. Everything else remains the same, and the security is just as perfect.

This all sounds good, but there are a few problems. The length of the key sequence is equal to the length of the message. This might be suitable for a few short messages, but this will never work for a 1.44 Mbps communications channel. However, you can store 650 megabytes worth of random bits on a CD-ROM. This would make a perfect one-time pad for certain low-bandwidth applications, although then you have to deal with the problem of storing the CD-ROM when it is not in use and then destroying it once it has been completely used.

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, the message won't make any sense. On the other hand, if some bits are garbled during transmission, only those bits will be decrypted incorrectly.

One-time pads still have their applications in today's world, primarily for ultrasecure 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; it doesn't matter how many people may still be working on the problem half a century later with unimaginable machines and techniques. 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 (as long as the one-time pads used to generate the messages have been destroyed).

1.3 LARGE NUMBERS

Throughout this book use some large numbers to describe various cryptographic algorithms. It's easy to lose sight of these numbers and what they actually mean. Table 1.1 gives physical analogues for the kinds of numbers used in cryptography.

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

Odds of being killed in an automobile accident (in the U.S. per year)	1 in 5600 (2^{-12})
Odds of being killed in an automobile accident (in the U.S. per lifetime)	1 in 75 (2^{-6})
Odds of drowning (in the U.S. per year)	1 in 59,000 (2^{-16})
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
If the Universe Is Closed:	
Total Lifetime of Universe	10^{11} (2^{37}) years
	10^{18} (2^{61}) seconds
If the Universe Is Open:	
Time Until Low-Mass Stars Cool Off	10^{14} (2^{47}) years
Time Until Planets Detach from Stars	10^{15} (2^{50}) years
Time Until Stars Detach from Galaxies	10^{19} (2^{64}) years
Time Until Orbits Decay by Gravitational Radiation	10^{20} (2^{67}) years
Time Until Black Holes Decay by the Hawking Process	10^{64} (2^{213}) years
Time Until All Matter Is Liquid at Zero Temperature	10^{65} (2^{216}) years
Time Until All Matter Decays to Iron	$10^{10^{26}}$ years
Time Until All Matter Collapses to Black Holes	$10^{10^{76}}$ years
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

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 178 road accidents per million people and an average lifespan of 75.4 years.

Cryptographic Protocols

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.) The types of problems that cryptography solves revolve around secrecy, distrustful and dishonest people, and privacy. You can learn all about algorithms and techniques, but these are merely interesting unless they solve problems. 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. Certainly one person can perform a series of steps to accomplish a task (e.g., baking a cake), but this is not a protocol. Finally, "designed to accomplish a task" means that the protocol must do 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:

1. Everyone involved in the protocol must know the protocol and all of the steps to follow in advance.
2. Everyone involved in the protocol must agree to follow it.

3. The protocol must be unambiguous; each step must be well defined and there must be no chance of a misunderstanding.
4. 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 parties, or messages from one party to another.

Cryptographic Protocols

A **cryptographic protocol** is a protocol that uses cryptography. The parties involved can be friends and trust one another implicitly, or they can be adversaries and not trust one another at all. Of course, 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. Cryptographic protocols that accomplish such goals have radically changed our ideas of what mutually distrustful parties can accomplish over a network.

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.

More and more, people are communicating over computer networks instead of communicating 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 (or even from one country 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 or her shuffle and deal? Would you mail the government your secret ballot without some assurance of anonymity?

It is naive to assume that the people on a computer network are going to be honest. It is naive to assume that the managers of a computer network are going to be honest. It is even naive to assume that the designers of a computer network were honest. Most will be honest, but it is the dishonest few we need to guard against. By formalizing protocols, we can examine ways in which dishonest parties can try to cheat and develop protocols that foil these cheaters.

In addition to formalizing behavior, protocols are useful because they abstract the process of accomplishing a task from the mechanism by which the task is

accomplished. A communications protocol between two computers is the same whether the computers are IBM PCs, VAX computers, or facsimile machines. This abstraction allows us to examine the protocol for good features 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 Company

To help demonstrate the 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. Bob will play the second part. 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 particular reason to complete the protocol and no particular allegiance to any of the people involved in the protocol. Trusted means that all people involved in the protocol accept that

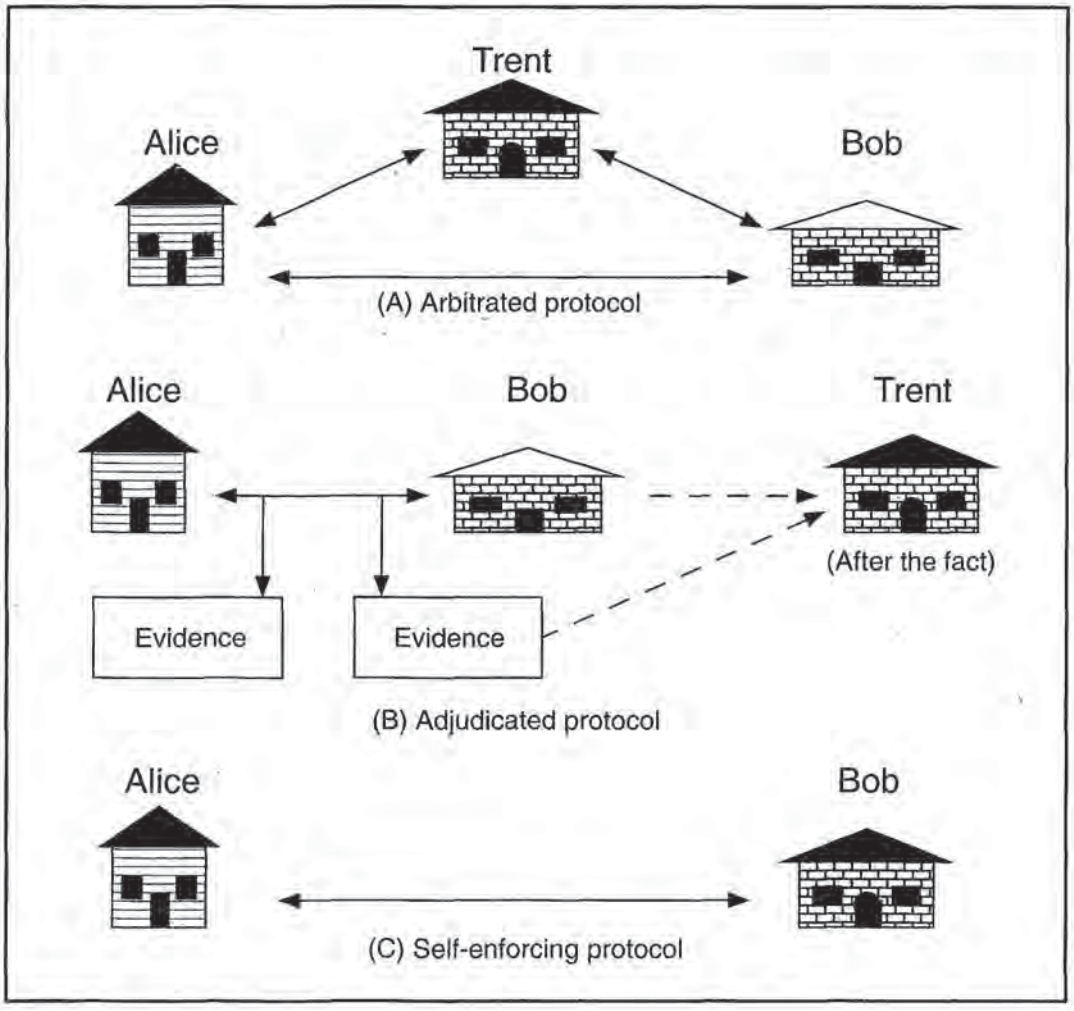


Figure 2.1
Types of protocols.

TABLE 2.1
Dramatis Personae

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

what is said is true, what is done is correct, and that his or her part of the protocol will be complete. 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 or not. 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.

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

(1) Alice gives the title and the keys 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.

(5) 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 and the keys to Alice.

In this protocol, Alice trusts the lawyer not to give Bob the title and the keys unless the check has cleared and to give them back to her if the check does not clear. Bob trusts the lawyer to hold the title and the keys until the check clears before giving them to him. The lawyer doesn't care if the check clears or not. He will do what he is supposed to do in either case.

In the example, the lawyer is playing the part of an escrow agent. Escrow agents often arbitrate financial transactions. Lawyers 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 and the keys to Bob.

(4) Bob gives the certified check to Alice.

(5) Alice deposits the check.

This protocol works because Alice trusts the banker's certification. Alice trusts the bank to hold her money 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 rosy picture 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 translating arbitrators into computers:

- It is easier to find and trust a neutral third party if you know who the party is, can see his face, or have a feeling that he is a real person. Two parties suspicious of each other are also likely to be suspicious of some 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.
- Arbitrators are potential bottlenecks in large-scale implementations of any protocol. Increasing the number of arbitrators in the implementation can mitigate this problem, but then the cost increases.
- 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 people 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 transaction 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 unless one of them hauls the other into court.

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

Nonarbitrated subprotocol (executed every time):

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

(2) Alice signs the contract.

(3) Bob signs the contract.

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

(1) Alice and Bob appear before a judge.

(2) Alice presents her evidence.

(3) Bob presents his evidence.

(4) The judge rules on the evidence.

The key difference between an adjudicator and an arbitrator (as I use the terms in this book) is that the adjudicator is not always necessary. If there is 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 involved parties to be honest; but if someone cheats, there is a body of data collected so that a disinterested third party could determine if someone cheated. In a good, adjudicated protocol, the adjudicator could also determine the cheater's identity. In real life, adjudicators are seldom called. The inevitability of detection discourages cheating, and people remain honest.

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. I'll do my best, though.

Attacks Against Protocols

Attacks against protocols can be directed against the cryptographic algorithms used in the protocol, the cryptographic techniques used to implement the algorithm (e.g., key generation), or the protocol iself. This section of the book discusses only the protocols. For the time being we will assume that the cryptographic algorithms and techniques are secure, and look only at attacks against the protocols themselves.

There are various ways people can try 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. 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 introduce new messages in the protocol, delete existing messages, substitute one message for another, destroy a communications channel, or alter stored information in a computer. These are called **active attacks,** because they require active intervention.

Passive attackers are concerned solely with obtaining information about the parties involved in the protocol. They do this by collecting the messages passing among various parties and attempting 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. There could even be many active attackers, all working together, each of them a legitimate system user. The part of the malicious active attacker will be played by Mallet.

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 system'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, an eavesdropper 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; there are an array of algorithms that are resistant (as far as we know) to whatever computing power Eve could bring to bear on the problem.

Eve isn't stupid, though. She knows that if she can listen in on steps (1) and (2), she's succeeded. She would know the algorithm and the key; she would know just as much as Bob. When the message comes across the communications channel in step (4), all she has to do is decrypt it herself.

This is why key management is such an important matter in cryptography. A good cryptosystem is one in which all the security is inherent in the key and none is inherent in the algorithm. 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; otherwise the message will no longer be secure. (Public-key cryptography solves this problem another way and will be discussed in Section 2.5.)

Mallet, 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. Mallet could also intercept Alice's messages and substitute ones of his own. If he also 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 Mallet didn't know the key, he could only create a replacement message that would decrypt to gibberish. Bob, thinking the message came from Alice, might conclude that either the network or Alice had some serious problems.

What about Alice? What can she do to disrupt the protocol? She can give a copy of the key to Eve. Now Eve can read whatever Bob says. Bob, who has no

idea that Eve has the key, thinks he is talking securely to Alice. He has no idea Eve is reprinting 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:

- If the key is compromised (stolen, guessed, extorted, bribed, etc.), then the adversary who had the key can decrypt all message traffic encrypted with that key. He or she can also pretend to be one of the parties and produce false messages to fool the other party. It is very important to change keys frequently to minimize this problem.
- Keys must be distributed in secret. They are more valuable than any of the messages they encrypt, since knowledge of the key means 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.
- 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. For example, 10 users need 45 different keys to talk with one another; 11 users need 55 different 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 **one-way functions** is central to public-key cryptography. While not a protocol in itself, one-way functions are a fundamental building block for most of the protocols described in this book.

A one-way function is a function that is relatively easy to compute but significantly harder to undo or 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" means, in effect, that it would take millions of years to compute the function even if all the computers in the world were assigned to the problem.

Taking a watch apart is a good example of a one-way function. It is easy to smash a watch into hundreds of tiny pieces. However, it's not easy to put all of those tiny pieces back together into a functional watch.

This sounds accurate, but in fact it isn't demonstratively true. If we are being strictly mathematical, there is no proof that one-way functions exist, nor is there any real evidence that they can be constructed [138,322,366,393]. Even so, there are many functions that look and smell one-way: we can compute them efficiently and, as of now, know of no easy way to reverse them. For example, x^2 is easy to compute, but $\sqrt{x}$ 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 9.2.

So, what good are one-way functions? I 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 the friend to read the message. Observe how impressed your friend is with the one-way function.) For encryption, we need something called a **trap-door one-way function.** (There are cryptographic applications for one-way functions; see Section 3.2.)

A **trap-door one-way function** is a special type of one-way function with a secret trap door. It is easy to compute in one direction and hard to compute in the other direction. But, if you know the secret, you can easily compute the function in the other direction. That is, it is easy to compute $f(x)$ given x, and hard to compute x given $f(x)$. However, there is some secret information, y, such that given $f(x)$ and y it is easy to compute x. In our watch example, the secret information might be a set of assembly instructions for the watch.

A mailbox is a good example of a trap-door one-way function. Anyone can easily put mail into the box; just open the slot and drop it in. Putting mail in a mailbox is a public activity. Opening the mailbox is not a public activity. It's hard; you would need welding torches or other tools. However, if you have the secret (the key or the combination), it's easy to open the mailbox. Public-key cryptography is a lot like that.

2.4 ONE-WAY HASH FUNCTIONS

A **one-way hash function** has many names: **compression function, contraction function, message digest, fingerprint, cryptographic checksum, data integrity check (DIC), manipulation detection code (MDC), message authentication code (MAC),** and **data authentication code (DAC).** 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 an input string and converts it to a fixed-size (often smaller) output string. A simple hash function would be a function that takes an input string and returns a byte consisting of the XOR of all the input bytes. The whole point here is to fingerprint the input string: to produce a value that can indicate whether a candidate string is likely to be the same as the input string. 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 equality.

A one-way hash function is a hash function that is also a one-way function: it is easy to compute a hash value from an input string, but it is hard to generate a string that hashes to a particular value. The hash function in the previous paragraph 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 particular one-way hash function may return values on the order of 128 bits long, so that there are 2^{128} possible hashes. The number of trials required to find a random string with the same hash value as a given string is 2^{128}, and the number of trials required to find two random strings having the same (random) hash value is 2^{64}.

There are two primary types of one-way hash functions: those with a key and those without a key. One-way hash functions without a key can be calculated by anyone; the hash is solely a function of the input string. One-way hash functions with a key are a function of both the input string and the key; only someone with the key can calculate the hash value. (It's the same as calculating the one-way hash and then encrypting it.)

Algorithms specifically designed to be one-way hash functions have been developed (see Chapter 14). These are **pseudo-random** functions; any hash value is equally likely. The output is not dependent on the input in any discernible way. A single change in any input bit changes, on the average, half the output bits. Given a hash value, it is computationally unfeasible to find an input string 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 her to send it to you, then ask her for the hash value. If she sends you the correct hash value, then it is almost certain that she has that file. Normally, you would use a one-way hash function without a key, so that anyone can verify the hash. If you only want the recipient to be able to verify the hash, then use a one-way hash function with a key.

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 [299]. They described **public-key cryptography.** Instead of one key, there are two different keys: one public and the other private. Moreover, it is computationally unfeasible to deduce the private key from the public key. Anyone with the public key (which, presumably, is public) can encrypt a message but not decrypt it. Only the person with the private key can decrypt the message. It is as if someone cut a mail slot into the cryptographic safe. Anyone can slip messages into the slot, but only someone with the private key can open the safe and read the messages.

Mathematically, the process is based on the trap-door one-way functions discussed above. 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 trap door, 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 then decrypts Alice's message using his private key.

Notice how public-key cryptography solves the primary problem with symmetric cryptosystems: by distributing the key. With a conventional cryptosystem, Alice and Bob have to agree on the same key. Alice could choose one at random, but she still has to get it to Bob. She could hand it to him sometime beforehand, but that requires foresight. She could send it to him by registered mail, but that takes time. With public-key cryptography, there is no problem. 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.

Attacks Against Public-Key Cryptography

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

The obvious way to exchange a 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 access by anyone except Trent; otherwise Mallet could substitute a key of his choosing for Alice's. After he did that, Bob couldn't read his messages, but Mallet could.

Even if the public keys are stored in a secure database, Mallet could still substitute one for another during transmission. To prevent this, Trent can sign each of the public keys 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, the 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 Mallet, only more difficult. Alice still has the KDC's public key stored somewhere. Mallet has to substitute that key for his own public key, corrupt the database, and substitute the

valid keys with his own 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 Mallet goes to enough trouble. As mentioned earlier, this will be discussed in minute detail in Section 3.1.

Hybrid Cryptosystems

Public-key algorithms are significantly slower than symmetric algorithms; symmetric algorithms are generally at least 1000 times faster than public-key algorithms. In most practical implementations public-key cryptography is used to secure and distribute session keys, and those session keys are used with symmetric algorithms to secure message traffic [499]. This is sometimes called a **hybrid cryptosystem** (see Section 3.1).

2.6 DIGITAL SIGNATURES

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

1. The signature is unforgeable. The signature is proof that the signer deliberately signed the document.
2. The signature is authentic. The signature convinces the document's recipient that the signer deliberately signed the document.
3. The signature is not reusable. The signature is part of the document, and an unscrupulous person cannot move the signature to a different document.
4. The signed document is unalterable. After the document is signed, it cannot be altered.
5. The signature cannot be repudiated. The signature and the document are physical things. The signer cannot later claim that he or she didn't sign it.

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

We would like to do this sort of thing on computers, but there are problems. First, bit streams are easy to copy. Even if a person's signature were difficult to forge (a graphical image of a written signature, for example), it is easy to move a valid signature from one document to another document. The mere presence of such a signature means nothing. Second, documents 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, a statement that he has received this message from Alice, and a copy of Alice's encrypted message. He 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 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 send the message to Bob.
2. This signature is authentic. Trent is a trusted arbitrator, and Trent knows that the message came from Alice. Trent's certification is proof enough.
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 notice that it did not match the encrypted message that Bob gave him. Bob, of course, could not produce an encrypted message that does match because he does not know K_A.
4. The signed document is unalterable. Were Bob to try to alter the document after receipt, Trent could prove foul play in exactly the same manner described above.
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 the 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 re-encrypts the bundle with the secret key he shares with Carol, K_C, and sends it to Carol.

(4) 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 has to 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 is going to be 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; even if he makes 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 gets out, or if someone manages to modify his code, everyone's signatures would be completely useless. False documents purported to be signed years ago could appear. Chaos would result. This might work in theory, but it doesn't work very well in practice.

Signing Documents with Public-Key Cryptography

There are public-key algorithms that can be used for digital signatures. In some algorithms—RSA (see Section 12.4) is an example—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 (see Section 13.5) is an example—there is a separate algorithm for digital signatures that cannot be used for encryption. This idea was first invented by Diffie and Hellman [299] and further expanded and elaborated on in other texts [723,749,570,724].

The basic protocol is simple:

(1) Alice uses a digital signature algorithm with her private key to sign the message.

(2) Alice sends the document to Bob.

(3) Bob uses the digital signature algorithm with Alice's public key to verify the signature.

This protocol is far better than the previous one. Trent is not necessary; Alice and Bob can do it by themselves. Bob does not even need Trent to resolve disputes: if he 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 unforgeable; only Alice knows her private key.
2. The signature is authentic; when Bob verifies the message with Alice's public key, he knows that she signed it.
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, it 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 signature and the document together. This isn't exciting 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 forever.

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 fifteen 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 encrypt long documents. To save time, digital signature protocols are often implemented with one-way hash functions [246,247]. 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 signs 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 verifies the signed hash with Alice's public key and compares it with the hash he generated. If they match, the signature is valid.

Speed increases drastically, and, since the chances of two different documents having the same 160-bit hash are only one in 2^{160}, anyone can safely equate a signature of the hash with a signature of the document. If a non-one-way hash function were used, it would be an easy matter to create multiple documents that hashed to the same value, so that anyone signing a particular document would be duped into signing a multitude of documents. This protocol cannot work without one-way hash functions.

This protocol has other benefits. First, the signature has to 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 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 3.8.)

Algorithms and Terminology

There are many digital signature algorithms. All of them are public-key algorithms: there is some secret information that only allows someone who knows the information to sign documents, and there is some public information that allows everyone to verify documents. 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. Additionally, there are often implementation differences. 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 message with the corresponding public key is:

$$V_K(M)$$

The bit string attached to the document when signed (in the above 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. In the first option, Alice and Bob sign separate copies of the document itself. The resultant message would be twice the size of the original document—this is not optimal. In the second option, Alice would sign the document first and then Bob would sign Alice's signature. This works, except it is impossible to verify Alice's signature without also verifying Bob's.

If a one-way hash of the document is signed instead of the document itself, multiple signatures are easy:

(1) Alice signs the document.

(2) Bob signs the document.

(3) Bob sends his signature to Alice.

(4) Alice or Bob sends the document, her signature, and his 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.

Cheating with Digital Signatures

Alice can cheat with digital signatures, and there's nothing that can be done about it. She wishes to 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 some public place, or just pretends to do either of the above. Now, anyone who finds the key can pretend to be Alice and sign documents. 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. 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 one hears so much about private keys buried in tamper-resistant modules—so that Alice can't get at hers and abuse it.

Although there is nothing that 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 prevent her from paying Bob for the junker he sold her yesterday and in the process invalidate her year-old mortgage.) The solution is for the receiver of a signed document to have

it timestamped by an arbitrator. This timestamp proves that the document was signed at a given time. Now, when Alice either pretends to lose or actually loses her key, only documents signed after she reports the loss are considered invalid. This is similar to the rules about reporting a stolen credit card.

Applications of Digital Signatures

One of the earliest proposed applications of digital signatures was to facilitate the verification of nuclear test ban treaties [817]. The United States and the former Soviet Union can put seismometers on each other's soil to monitor each other's nuclear tests. The problem is that the monitoring nation must assure itself that the host nation is not tampering with the data from the monitoring nation's seismometers. Simultaneously, the host nation wants to assure itself that the monitor is sending only the specific information needed for monitoring. Conventional authentication techniques can solve the first problem, but only digital signatures can solve both problems. The host nation can read, but not alter, data from the seismometer; and the monitoring nation knows that the data has not been tampered with.

2.7 DIGITAL SIGNATURES WITH ENCRYPTION

By combining digital signatures with public-key cryptography, we can develop a protocol that combines the security of encryption with the authenticity of digital signatures. For example, think of a signed letter in an envelope: the digital signature provides proof of authorship, and encryption 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.

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

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 in the next section.

Resending the Message as a Receipt

Consider an implementation of this protocol, with the additional feature of confirmation messages. Whenever someone receives a message, he or she sends it back to the sender as a confirmation of receipt.

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

$$E_B(S_A(M))$$

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

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

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

$$E_A(S_B(M))$$

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

If the same algorithm is used for both encryption and digital signatures there is a possible attack [305]. In these cases, the digital signature operation is the inverse of the encryption operation: $V_X = E_X$ and that $S_X = D_X$.

Assume that Mallet 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 (Mallet). Bob thinks that it is a legitimate message from Mallet, so he decrypts the message with his private key and then tries to verify Mallet's signature by decrypting it with Mallet'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 Mallet a receipt:

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

Now, all Mallet 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! Mallet has M.

It is not unreasonable to imagine that Bob may automatically send Mallet a receipt. This protocol may be embedded in his communications software, for example, and send a receipt automatically upon receipt. 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 Mallet to send Bob a different message from the one he eavesdropped on. It is important never to sign arbitrary messages from other people or to decrypt arbitrary messages and give the results to other people.

Foiling the Resend Attack

The above attack works because the encrypting operation is the same as the signature-verifying operation, and the decryption operation is the same as the signature operation. If the protocol used even a slightly different operation for encryption and digital signatures, this would be avoided. Digital signatures with one-way hash functions solve the problem (see Section 2.6); so does using different keys for each operation; so do timestamps, which make the incoming message and the outgoing message different.

In general, then, this protocol is perfectly secure:

(1) Alice signs a message.

(2) Alice encrypts the message and signature with Bob's public key (using a different encryption algorithm than she used for the signature) and sends it to Bob.

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

(4) Bob verifies Alice's signature.

It is possible to modify the protocol so that Alice encrypts the message before signing it. While this might be suitable for some circumstances, when an intermediate party would need to verify the signature without being able to read the message, in general it is better to encrypt everything. Why give Eve any information at all?

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. Unfortunately, those random number generators are almost definitely not cryptographically secure, and probably not even very random. Most of them are embarrassingly bad.

Random sequence 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 sequence generator and you start getting weird correlations and strange results [686,691]. If security depends on your random number generator, 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. Knuth quotes John von Neumann as saying: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" [488]. 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. There are only a finite number of states in which a computer can exist (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 sequence generator on a computer (at least, on a Turing 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 sequence generator requires some random input; a computer can't provide that.

2.8.1 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. The sequence's period should be long enough so that a finite sequence of reasonable length, i.e., one that is actually used, is not periodic. That is, if you need a billion random bits, don't choose a sequence generator that repeats after only sixteen thousand bits. These relatively short, non-periodic 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 runs of zeros and the other half should be runs of ones; half the runs should be of length one, one quarter of length two, one eighth of length three, etc. 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 [488].)

A lot of effort has gone into producing good pseudo-random sequences on computers. 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 biggest applications.

The problem is still those weird correlations and strange results. Every deterministic 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.

2.8.2 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 random,** it must have this additional second property:

2. **It is unpredictable.** It must be computationally unfeasible 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.

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.

2.8.3 Real Random Sequences

Now we're meandering 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 is that randomness preserved when brought to the macroscopic world of computer chips and finite-state machines?

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

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

Additionally, real random sequences cannot be compressed. Cryptographically secure pseudo-random sequences cannot, in practice, be compressed.

The output of a generator with these properties will be good enough for a one-time pad, key generation, and any other cryptographic properties for which one might want it.

Basic Protocols

3.1 KEY EXCHANGE

A common cryptographic technique is to encrypt each individual conversation between two people with a separate key. This is called a **session key,** because it is used for only one particular communications session. How this common session key gets into the hands of the conversants can be a complicated matter.

Key Exchange with Symmetric Cryptography

(1) Alice calls a Key Distribution Center (KDC) and requests a session key to communicate with Bob.

(2) The KDC generates a random session key. It encrypts two copies of it: one in Alice's key and the other in Bob's key. The KDC also encrypts information about Alice's identity with Bob's key. The KDC sends both copies to Alice.

(3) Alice decrypts her copy of the session key.

(4) Alice sends Bob his copy of the session key and the identity information.

(5) Bob decrypts his copy of the session key and the identity information. He uses the identity information to determine who Alice is. (Presumably Alice knows who Bob is; she called him and only he can decrypt the key.)

(6) Both Alice and Bob use this session key to communicate securely.

This protocol works, but it relies on the absolute security of the KDC. If Mallet corrupts the KDC, the whole network is compromised. He has all of the secret

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

Key Exchange with Public-Key Cryptography

The hybrid cryptosystem was discussed in Section 2.5.

(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.

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

Key Exchange with Public-Key Cryptography Using a Public-Key Database

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 message to Bob even if she has never met him:

(1) Alice gets Bob's public key from a central database.

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

(3) Bob then decrypts Alice's message using his private key.

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

Man-in-the-Middle Attack

While Eve cannot do better than try to break the public-key algorithm or attempt a ciphertext-only attack on the ciphertext, Mallet can decrypt messages between Alice and Bob. Mallet 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. Mallet 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. Mallet intercepts this key and sends Bob his own public key.

(2) Bob sends Alice his public key. Mallet intercepts this key and sends Alice his own public key.

(3) When Alice sends a message to Bob, encrypted in "Bob's" public key, Mallet 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, Mallet 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 and Bob's public keys are stored on a database, this attack will work. Mallet can intercept Alice's database inquiry, and substitute his own public key for Alice's. He can do the same to Bob. He can break into the database surreptitiously and substitute his key for Alice's and Bob's. The next day he simply waits for Alice and Bob to talk with each other. Then he 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 Mallet is quick and 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.

The easiest way to protect against this is for Alice and Bob to confirm that they are using the same key. If they can communicate via voice (or another way that allows them to unambiguously identify each other), they can each read a one-way hash of their key to the other. Of course, this isn't always possible.

Interlock Protocol

The **interlock protocol,** invented by Ron Rivest and Adi Shamir [748], 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 (for example, every other bit) could be sent in each half message.
- Decryption of the message could be dependent on an initialization vector (see Section 8.1.3), which would be sent with the second half of the message.
- The first half of the message could be a one-way hash function of the encrypted message (see Section 2.4), and the encrypted message itself could be the second half.

To see how this causes a problem for Mallet, 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 has to 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.

Mallet 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. A KDC 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 the KDC's signature. Now they know that the public key belongs to that other person. The key exchange protocol can then proceed.

Mallet 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 it isn't signed by the KDC. 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 also uses a KDC, but the risk of compromising the KDC is much less. If Mallet breaks into the KDC, all he gets is the KDC'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 be able to read the traffic, Mallet has to be able to impersonate a user on the network and trick legitimate users into encrypting messages with his phony public key.

Mallet can launch that kind of attack. With the KDC'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 Mallet has to be a powerful attacker. Intercepting and modifying messages is a lot more difficult than passively sitting on a symmetric network and reading messages as they go by. Active wiretaps are much harder to put in place and much easier to detect. On a broadcast channel, such as a radio network, it is almost impossible to replace one message with another.

Key and Message Transmission

There is no reason for Alice and Bob to complete the key-exchange protocol before exchanging messages. In this protocol, Alice sends Bob the message, M, without any previous key-exchange protocol:

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

$$E_K(M)$$

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

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

$$E_B(K)$$

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

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

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

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

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

This 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.

(5) Only Bob, Carol, and Dave can decrypt the K key 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 computer (or an automatic teller, or a telephone banking system, for example), how does the computer know who she is? How does the computer know she is not someone else trying to falsify her identity? Traditionally, passwords solve this problem. Alice enters her password, and the computer confirms that it is correct. Both Alice and the computer know this secret piece of knowledge, and the computer requests it from Alice every time she tries to log in.

What several cryptographers realized is that the computer does not need to know the passwords; the computer just has to be able to differentiate valid passwords from invalid passwords. This is easy with one-way functions [320,621,715,879]. Instead of storing passwords, the computer stores one-way functions of the passwords.

(1) Alice sends the computer her password.

(2) The computer performs a one-way function on the password.

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

Since the computer no longer stores a table of everybody's valid password, the threat of someone breaking into the computer 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

Even a file of passwords encrypted with a one-way function is vulnerable. In his spare time, Mallet compiles a list of the 100,000 most common passwords. He operates on all 100,000 of them with the one-way function and stores the results.

If each password is about eight bytes, the resulting file will be no more than 800K; it will fit on a single floppy disk. Now, Mallet steals the encrypted password file. He compares the encrypted password with his file of encrypted possible passwords and sees what matches.

This is a **dictionary attack,** and it's surprisingly successful (see Section 7.2.1). **Salt** is a way to make it more difficult.

Salt is a random string that is concatenated with the password 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 the database. If the number of possible salt values is large enough, this practically eliminates a dictionary attack against commonly used passwords. This is a simple attempt at an **Initialization Vector** (see Section 8.1.3).

A lot of salt is needed. Most UNIX systems use only twelve bits of salt. Even with that, Daniel Klein developed a password-guessing program that cracks 21% of the passwords on a given system in about a week [482]. 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% of passwords on a given system can be broken with this list [346].

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 obscures people who have the same password on multiple machines, but doesn't make poorly chosen passwords any better.

User Identification with Public-Key Cryptography

Even with salt, the above protocol has serious security problems. First, when Alice types her password into the system, anyone who has access to her data path can read it. She might be accessing her computer through a convoluted transmission path that passes through four industrial competitors, three foreign countries, or two forward-thinking universities, and a partridge in a pear tree. Any one of those points can have an Eve listening to Alice's log-in sequence. Second, anyone with access to the processor memory of the computer system can see the password before the system encrypts it.

Public-key cryptography can solve this problem. The host computer keeps a file of every user's public key; all users keep their own private key. Here is a naive 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 identity.

(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 the user. 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.

In general, it is foolish to encrypt random strings sent by another party; attacks similar to the one discussed in Section 12.4 can be mounted.

In general, secure proof-of-identity protocols take the following form:

(1) Alice performs a computation based on some random numbers and her secret key and sends the result to the host.

(2) The host sends Alice a different random number.

(3) Alice makes a computation based on the random numbers (both the ones she generated and the one she received from the host) and her secret key and sends the result to the host.

(4) The host does a 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 does not trust Alice, then Alice will then require the host to prove its identity.

Step (1) might seem unnecessary and confusing, but it is required to prevent attacks against the protocol. Section 12.7 and Section 13.1 mathematically describe several algorithms and protocols. See also [527].

Mutual Authentication Using the Interlock Protocol

Alice and Bob are two users who want to authenticate each other. Each of them has a password: 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 P_B and verifies that it is correct.

(5) Bob decrypts P_A and verifies that it is correct.

The problem is that Mallet can launch a successful man-in-the-middle attack (see Section 3.1):

(1) Alice and Bob trade public keys. Mallet 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. Mallet 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. Mallet 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.

From what Alice and Bob see, nothing is different. However, Mallet knows both P_A and P_B.

Davies and Price describe how the interlock protocol can defeat this attack [249]:

(1) Alice and Bob trade public keys.

(2) Alice encrypts P_A with Bob's public key and sends half of it to him.

(3) Bob encrypts P_B with Alice's public key and sends half of it to her.

(4) Alice sends the other half of encrypted P_A to Bob.

(5) Bob combines the two halves, decrypts P_A, and verifies that it is correct.

(6) Bob sends the other half of encrypted P_B to Alice.

(7) Alice combines the two halves, decrypts P_B, and verifies that it is correct.

Steve Bellovin and Michael Merritt discuss ways to attack this protocol [60]. If Alice is a user and Bob is a host, Mallet can pretend to be Bob, complete steps (1) through (5) of the protocol with Alice, and then drop the connection. True artistry demands Mallet do this by simulating line noise or network failure, but the final result is that Mallet has Alice's password. He can then connect with Bob and complete the protocol. Voilà. Mallet now has Bob's password.

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 [60].

SKID

SKID2 and SKID3 are secret-key identification protocols developed for RACE's RIPE project [727] (see Section 18.8). They use a keyed one-way hash function (a MAC) to provide security, and both assume that both Alice and Bob share a secret key, K.

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

(1) Alice chooses a random number, *RA*. (The RIPE document specifies a 64-bit number.) She sends it to Bob.

(2) Bob chooses a random number, *RB*. (The RIPE document specifies a 64-bit number.) He sends Alice:

$RB,HK(RA,RB,\text{Bob})$

HK is the MAC. (The RIPE document suggests the RIPE-MAC function—see Section 14.14.) Bob is Bob's name.

(3) Alice computes $HK(RA,RB,\text{Bob})$ 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:

$HK(RB,\text{Alice})$

Alice is Alice's name.

(5) Bob computes $HK(RB,\text{Alice})$, and compares it with what he received from Bob. If the results are identical, then Bob knows that he is communicating with Alice.

3.3 AUTHENTICATION AND KEY EXCHANGE

These protocols solve a general computer problem: Alice and Bob are sitting on other ends of the network. They want to talk securely. How can Alice and Bob exchange a secret key and at the same time each be sure they are talking to the other and not to Mallet?

Wide-Mouth Frog

The Wide-Mouth Frog protocol [180] is probably the simplest symmetric key-management protocol that uses a trusted server. Both Alice and Bob share a secret key with Trent. These keys are themselves distributed secretly and authentically over some external channel, which we shall assume to be secure. The keys are just used for key distribution and not to encrypt any actual messages between users. Just by using two messages, a session key is transferred from Alice to Bob:

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

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

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

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

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

Yahalom

This protocol is by Yahalom [180]. Like the previous protocol, both Alice and Bob share a secret key with Trent.

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

$$A,R_A$$

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

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

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

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

(4) Alice decrypts the 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.

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

(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 and Schroeder

This protocol, invented by Needham and Schroeder [645], also uses symmetric cryptography and Trent.

(1) Alice sends a message to Trent consisting of her name, A, Bob's name, B, and some random value R_A.

$$(A,B,R_A)$$

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

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

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

$$E_B(K,A)$$

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

$$E_K(R_B)$$

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

$$E_K(R_B-1)$$

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

All of this fussing around with R_A and R_B and R_B-1 is to guarantee that there are no replay attacks. The presence of R_A in step (3) assures Alice that Trent's message is legitimate and not a replay of an old 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 earlier executions of the protocol.

The biggest security problem with this protocol is that old session keys are valuable. If Mallet gets access to an old K, he can launch a successful attack [272]. 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) Mallet sends Bob the following message:

$$E_B(K,A)$$

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

$$E_K(R_B)$$

(3) Mallet 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, Mallet has Bob convinced that he is Alice.

A stronger protocol, using timestamps, can defeat this attack [272]. This protocol requires a secure and accurate system clock, not a trivial problem in itself. This modified protocol is the basis for the Kerberos authentication protocols (see Section 17.4).

There are even more drastic consequences if K_A is ever compromised. Mallet can use it to obtain session keys to talk with Bob (or anyone else he wishes to talk to). Even worse, Mallet can continue to do this even after Alice changes her key [47].

Needham and Schroeder attempted to correct these problems in a modified version of their protocol [646]. Their new protocol is essentially the same as the Otway-Rees protocol, published in the same issue of the same journal.

Otway-Rees

Otway and Rees's protocol also uses symmetric cryptography [684]. As usual, Trent shares a separate key with everyone on the network.

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

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

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

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

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

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

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

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

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 has been implemented and is discussed in detail in Section 17.4. The basic Kerberos Version 5 protocol is as follows: 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, A, and Bob's identity, B.

$$A,B$$

(2) Trent generates a message with a timestamp, T, a lifetime, L, a random session key, K, 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 this in the key he shares with Alice:

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

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

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

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

$$E_K(T+1)$$

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

SPX

The SPX protocols, developed at Digital Equipment Corporation, also provide for mutual authentication and key exchange [851,850]. Unlike the other protocols, SPX uses both public-key and symmetric cryptography. Alice and Bob each has a private key. Trent has signed copies of their public keys.

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

$$B$$

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

$$S_T(K_B)$$

(3) Alice verifies Trent's signature to confirm that the key she received is actually Bob's public key. She generates a random secret key, K, and a random public-key/private-key key pair: K_P. She encrypts the time, T_A, with K. Then she signs a key lifetime, L, her identification, A, and K_P with

her private key, K_A. Finally, she encrypts K with Bob's public key, and signs it with K_P. She sends all of this to Bob.

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

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

$$A$$

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

$$S_T(K_A)$$

(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, T_B, 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.

DEC has a working implementation of the SPX protocols. Additional information can be found in [18].

Other Protocols

There are many other protocols in the literature. The CCITT X.509 protocols are discussed in Section 17.6. KryptoKnight is discussed in Section 17.5. Another protocol, Encrypted Key Exchange, is discussed in Section 16.2.

3.4 MULTIPLE-KEY PUBLIC-KEY CRYPTOGRAPHY

In public-key cryptography, there are 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 [134]. Imagine a variant of public-key cryptography with three keys: K_A, K_B, and K_C. Alice has the first, Bob the second, and Carol the third. In addition, Dave has both K_A and K_B. Ellen has both K_B and K_C. Frank has both K_C and K_A. Any number of keys can be used to encrypt a message. The remaining keys are required to decrypt that message.

Alice can encrypt a message with K_A so that Ellen, with K_B and K_C, can decrypt it, as 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. No other pairs can communicate. This is summarized in Table 3.1:

TABLE 3.1
Three-Key Message Encryption

ENCRYPTED WITH KEYS:	MUST BE DECRYPTED WITH KEYS:
K_A	K_B and K_C
K_B	K_A and K_C
K_C	K_A and K_B
K_A and K_B	K_C
K_A and K_C	K_B
K_B and K_C	K_A

This can be extended to n keys. If a certain 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 three operatives out in the field: Alice, Bob, and Carol. 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. The first option requires a lot more communications traffic; the second requires a lot more keys.

Multiple-key cryptography is much easier. 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 one can appreciate the ease of this scheme. If you want to send messages to subsets of those operatives, you have three choices: you can share a key with each operative (100 keys total) and send individual messages. You can distribute $2^{100}-1$ keys to account for every possible subset. Or, you can use this scheme; it works with only one encrypted message and 100 different keys.

There are other techniques for message broadcasting. These are discussed in Section 16.4.

3.5 SECRET SPLITTING

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

This calls for **secret splitting.** There are ways to take a message and divide it up into pieces [338]. Each piece by itself means nothing, but put them together and the message appears. If each employee has a piece of the recipe, then only together can they make the sauce. If any employee resigns with one piece of the recipe, the information is useless by itself.

The simplest sharing problem 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 P.

$$M \text{ XOR } R = P$$

(3) Trent gives P to Alice and R 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:

$$P \text{ XOR } R = M$$

This technique, if done properly, is absolutely secure. Each piece, by itself, is absolutely worthless. Only together can Alice and Bob reconstruct the message.

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 1.2.4 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 pieces to generate P:

$$M \text{ XOR } R \text{ XOR } S \text{ XOR } T = P$$

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

And here are Alice, Bob, and Carol reconstructing the message:

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

$$P \text{ XOR } R \text{ XOR } S \text{ XOR } T = M$$

This is an adjudicated protocol. Trent has absolute power and can do whatever he wants. He can hand out gibberish and claim that they are valid pieces 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 he can fire Bob. But since this is Trent's secret to divide up, this isn't a problem.

However, there is a problem with this protocol: if any of the pieces gets lost, so does the message. So 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, or Dave. All Alice, Bob, and Dave, combined, know is the length of the message—nothing more. Her piece is as critical to the message as every other piece combined. The next section shows how to fix this problem.

3.6 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 an easy problem 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 an entire city.)

We can get even more complicated. Maybe the general and two colonels can launch the missile, but if the general is busy playing golf then five colonels have to get together. 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, but 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. At its simplest level, you can take any message (e.g., a secret recipe, launch codes, or even a laundry list) and divide it into n pieces, called **shadows,** 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.

Threshold schemes are far more versatile than that. Any sharing scenario can be modeled using this scheme. You can divide a message among the people in your building so that you need seven people from the first floor and five people from the second floor, unless there is someone from the third floor, 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, 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, and so on.

This idea was invented independently by Adi Shamir [788] and G. R. Blakley [117]. There are several different algorithms, discussed in Section 16.5.

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

Scenario 2: Colonels Alice and Bob are sitting in the bunker with Mallet. Mallet 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. "Ha, Ha!" shouts Mallet. "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 Mallet, who is again disguised. The same message comes in from the president, and everyone reveals their shadows. Mallet reveals his shadow only after he has heard the other three. Since the other three reconstruct the secret, he quickly creates 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 16.5.

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, except there is 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. See [443] for further details about this protocol.

3.7 CRYPTOGRAPHIC PROTECTION OF DATABASES

The membership database of a professional 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 tea. On the other hand, if you distribute copies of 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 in such a way 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 [337,336], is straightforward. Choose a one-way hash function and a symmetric encryption algorithm. Each record of the database consists of 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.

Searching for a member with 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 there is a person in the database with the last name. 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 [337] the authors use this system to protect a natural-language dictionary of 6000 Spanish verbs. They report minimal performance degradation due to the encryption. Additional complications in [336] handle searching 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 salesperson to reconstruct the membership database by trying every possible last name. If he or she has a telephone database, that can be used. This might take a few weeks of dedicated computer crunching, but it can be done. Still, it makes the job harder.

Another approach, in [120], allows statistics to be compiled on encrypted data.

3.8 TIMESTAMPING SERVICES

There are many situations in which people need to certify that a document existed on a certain date. Think about a copyright or patent dispute: the party that can produce the earliest copy of the disputed work wins the case. There are many ways to do this with paper documents. Notaries can sign documents; lawyers can safeguard copies. If a dispute arises, the notary or the lawyer provides testimony that the letter existed on a certain date. (Of course, it is easy to send an unsealed empty envelope to yourself and then fill it with a letter at a later date.)

In the digital world, this is far more complicated. There is no way to examine a digital document for signs of tampering. They 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 December 10th, 1942."

Stuart Haber and W. Scott Stornetta at Bellcore thought about the problem [411,412,48]. They wanted a digital timestamping protocol that has the following three properties:

- The data itself must be timestamped, without any regard to the physical medium on which it resides.
- It must be impossible to change even one 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 actual one.

Arbitrated Solution

This protocol uses Trent, who has a trusted timestamping service. Of course, Alice 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 called 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 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 databanks. Who knows how secure that databank is?

Second, the databank itself would have to be huge, since the storage requirements would be so great. And the bandwidth requirements to send large documents to Trent would be enormous.

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

Fourth, there might not be someone as honest as Trent to run the timestamping service. Maybe Alice is using Bob's Time Stamp and Taco Stand. There is nothing to stop Alice and Bob from 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 the 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 and 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 timestamp 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 the previously received timestamps. 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. Alice's request is thus sandwiched in time.

If A identifies Alice, the hash value that Alice wants timestamped is H_n, and the previous timestamp is T_{n-1}, then the protocol is:

(1) Alice sends Trent H_n and A.

(2) Trent sends back to Alice:

$$T_n = S_K(n,A,H_n,T_n; ID_{n-1},H_{n-1},T_{n-1},H(ID_{n-1},T_{n-1},H_{n-1}))$$

(3) 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.

(4) After Trent stamps the next document, he sends Alice the identification of the originator of that document: ID_{n+1}.

If someone challenges Alice's timestamp, she just contacts the originators of the previous and following documents: ID_{n-1} and ID_{n+1}. If their documents are called into question, they can get in touch with ID_{n-2} and ID_{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.

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, 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 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 ID_{n-1}'s timestamp. This problem could be alleviated by embedding the previous ten people's timestamps into Alice's, and then sending Alice the IDs of the next ten people. Alice has a greater chance of finding people who still have their timestamp.

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:

$$V_1, V_2, V_3 \ldots V_k$$

(2) Alice interprets each of these values as the ID 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 IDs as verifiers. Even if she makes trivial changes in her document in an attempt to construct a set of corrupt IDs, her chances of getting away with this are negligible. The hash function randomizes the IDs; 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 the 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 [48]. Based on an idea by Ralph Merkle [588], 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 the newspaper. This serves a function similar to sending the hash to random people in the distributed protocol. In fact, Bellcore has been doing that in every Sunday's *New York Times* since 1992.

Keys

7.1 KEY LENGTH

The security of a symmetric cryptosystem is a function of two things: the strength of the algorithm and the length of the key. The latter is easier to demonstrate.

Assume that the strength of the algorithm is perfect. This is extremely difficult to achieve in practice, but easy enough for this example. By perfect, I mean that there is no better way to break the cryptosystem other than to try every possible key; this is called a brute-force attack. If the key is 8 bits long, there are 2^8=256 possible keys. Therefore, it will take 256 attempts to find the correct key, with an expected number of attempts of 128. If the key is 56 bits long, then there are 2^{56} possible keys. Assuming a supercomputer can try a million keys a second, it will take 2000 years to find the correct key. If the key is 64 bits long, then it will take the same supercomputer almost 600,000 years to find the correct key among the 2^{64} possible keys. If the key is 128 bits long, it will take 10^{25} years. The universe is only 10^{10} years old, so 10^{25} years is probably long enough. With a 2048-bit key, a million million-attempts-per-second computers working in parallel will spend 10^{597} years finding the key. By that time the universe may have collapsed or expanded into nothingness.

Before you rush to invent a cryptosystem with an 8-Kbyte key, remember that there is another half to this strength equation: the algorithm must be so secure that there is no better way to break it than with a brute-force attack. This is not as easy as it might seem. Cryptography is a subtle art. Cryptosystems that look perfect are often extremely poor. Strong cryptosystems, with a couple of minor changes, can become weak. Two strong cryptosystems, if used together poorly, can counteract each other and produce an output that is easy to break. The warning to the amateur cryptographer is to have a healthy, almost paranoid, suspicion of any new

algorithm. It is best to trust algorithms because professional cryptographers have scrutinized them for years without cracking them, not because the algorithm's designer makes grandiose claims about its security.

This brings up another important point: the security of a cryptosystem should rest in the key, not in the details of the algorithm. If the strength of your new cryptosystem relies on the fact that the attacker does not know the algorithm's insides, 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 it is naive to think that someone won't disassemble your source code or reverse-engineer your algorithm. (The National Security Agency keeps their algorithms secret from outsiders, but they have the best cryptographers in the world working within their walls. Additionally, they discuss their algorithms with one another, relying on peer review to uncover any weaknesses in their work.)

Assume that some cryptanalysts have access to all the details of your algorithm. Assume they have access to as much ciphertext as they want and can mount an intensive ciphertext-only attack. Assume that they can mount a plaintext attack with as much data as needed. Even assume they can mount a chosen-plaintext attack. If your cryptosystem can remain secure even in the face of all that knowledge, then you've got the security you need.

That warning aside, there is still plenty of room in cryptography to maneuver. In reality, there are many situations in which the kind of security discussed here isn't really necessary. Most adversaries don't have the knowledge and computing resources of a major government, and even the ones who do probably aren't that interested in breaking your cryptosystem. If you're plotting to overthrow a major government, stick with the tried and true algorithms in Section 3. The rest of you, have fun.

Time and Cost Estimates for Brute-Force Attack

If you assume that a brute-force attack is the most efficient attack possible against an algorithm—a big assumption—then the key must be long enough to make the attack unfeasible. How long is that?

Two parameters determine the speed of a brute-force attack: the number of keys to be tested and the speed of each test. Most symmetric algorithms accept any fixed-length bit pattern as the key. DES has a 56-bit key; there are 2^{56} possible DES keys. (The key is actually 64 bits, but 8 of those bits are parity bits.) Some algorithms discussed in this book have a 64-bit key; there are 2^{64} possible keys for those algorithms. Others have a 128-bit key.

The speed at which each possible key can be tested is also a factor, but a less important one. For the purposes of this analysis, I will assume that each different algorithm can be tested in the same amount of time. The reality may be that one algorithm may be tested two, three, or even ten times faster than another. But since we are looking for key lengths that are millions of times more difficult to crack than would be feasible, small differences due to test speed are irrelevant.

Most of the debate in the cryptologic community about the efficiency of brute-force attacks has centered on the DES algorithm. In 1977, Whitfield Diffie and Martin Hellman [300] postulated the existence of a special-purpose DES cracking machine. This machine consisted of a million chips, each capable of testing a million keys per second. Such a machine could test 2^{56} keys in 20 hours. If built to attack an algorithm with a 64-bit key, it could test all 2^{64} keys in 214 days.

A brute-force attack is tailor-made for parallel processors. Each processor can test a subset of the keyspace. The processors do not have to communicate among themselves; the only communication required at all is a single message signifying success. There are no shared memory requirements. It is easy to design a machine with a million parallel processors, each working independent of the others.

Special-purpose chips capable of testing a million keys per second are not farfetched. Chips capable of testing four million keys per second are feasible using today's technology [367]:

> We estimate that such chips [capable of 32,000,000 encryption operations per second] will be feasible within five years, based upon the history of semiconductor fabrication over the past twenty years. Today, using state-of-the-art techniques, it should be possible to develop a chip from scratch capable of checking 4,000,000 keys per second. If we limit ourselves to proven technology, chips that checked two million keys per second are possible. Within ten years we should be able to design chips capable of checking 256,000,000 keys per second.

With an array of four thousand chips, each capable of checking 265 million keys per second, all possible 56-bit keys can be tested within 24 hours. An array of 800,000 chips can test all possible 64-bit keys in the same period. If your data needs to be secure for more than ten years, these key lengths are inadequate.

Table 7.1 gives the number of parallel processors required for a brute-force attack against a given key length and within a specified period, assuming different speeds of the individual processors.

These numbers tell only part of the story. If a cracking machine will definitely recover a key in a year, there is an 8% chance it will recover the key in a month. If the key is only changed once a month, then there is an 8% chance that an adversary will recover the key while it is still being used. If that key is being used to authenticate a financial transfer system, this is a serious problem. Even if an adversary learns a valid key ten minutes before it is changed, he can make some very profitable transactions.

Table 7.2 gives the probability of finding an arbitrary key in a given period, assuming an adversary has a brute-force machine. For example, an adversary who has a machine capable of finding a key in a month has a 0.14% chance of finding the key within an hour. This means that if you change your key every hour, then in any given month, the adversary has a 63% chance of finding one of the keys before it is changed. This 63% is independent of the frequency of the key change.

TABLE 7.1
Number of Processors Required for a Brute-Force Attack Against Different Algorithms Within a Given Time Period

(NUMBER OF ENCRYPTIONS PER SECOND)	NUMBER OF PROCESSORS REQUIRED TO BREAK A 56-BIT KEY IN: 1 YEAR	1 MONTH	1 WEEK	1 DAY
1 million	2,300	28,000	120,000	830,000
2 million	1,100	14,000	60,000	420,000
4 million	570	7,000	30,000	210,000
32 million	71	870	3,700	26,000
256 million	9	100	470	3,300
(NUMBER OF ENCRYPTIONS PER SECOND)	NUMBER OF PROCESSORS REQUIRED TO BREAK A 64-BIT KEY IN: 1 YEAR	1 MONTH	1 WEEK	1 DAY
1 million	590,000	7,100,000	3.1×10^{7}	2.1×10^{8}
2 million	290,000	3,600,000	1.5×10^{7}	1.1×10^{8}
4 million	150,000	1,800,000	7,600,000	5.3×10^{7}
32 million	18,000	220,000	950,000	6,700,000
256 million	2,300	28,000	120,000	830,000
(NUMBER OF (ENCRYPTIONS PER SECOND)	NUMBER OF PROCESSORS REQUIRED TO BREAK A 128-BIT KEY IN: 1 YEAR	1 MONTH	1 WEEK	1 DAY
1 million	1.1×10^{25}	1.3×10^{26}	5.6×10^{26}	3.9×10^{27}
2 million	5.4×10^{24}	6.6×10^{25}	2.8×10^{26}	2.0×10^{27}
4 million	2.7×10^{24}	3.3×10^{25}	1.4×10^{26}	9.9×10^{26}
32 million	3.4×10^{23}	4.1×10^{24}	1.8×10^{25}	1.2×10^{26}
256 million	4.2×10^{22}	5.1×10^{23}	2.2×10^{24}	1.5×10^{25}

Changing keys frequently limits the amount of damage an adversary can do with a single recovered key, but in many applications this is still an unacceptable risk.

The other variable governing the feasibility of a brute-force cracking machine is the cost. The analysis used here, which appears in [367], was created using the following assumptions:

TABLE 7.2
Probability of Recovering a Key Within a Given Time (Assuming the Capability to Launch a Brute-Force Attack)

(BRUTE-FORCE ATTACK SPEED)	1 HOUR	1 DAY	1 WEEK	1 MONTH	1 YEAR
1 Day	0.0417	1.0000	1.0000	1.0000	1.0000
1 Week	0.0060	0.1429	1.0000	1.0000	1.0000
1 Month	0.0014	0.0329	1.2308	1.0000	1.0000
1 Year	0.0001	0.0027	0.0192	0.0833	1.0000

> In arriving at an estimate of the cost of breaking [the algorithm] we have to make a number of rather broad simplifications. Based on the technology assessment . . . we assume that the cost of a given technology will decrease roughly as quickly as it is outdated. In other words, a chip costing $100.00 today will cost (in today's dollars) $7.83 five years from now. Exotic chips available in ten years through custom fabrication will still be too expensive to be practical, but today's exotic technologies will be dirt cheap. To be conservative, a cost of $25.00 per processor was assumed for the cheapest available technology, rising by a factor of ten for each "generation" of technology (roughly, five years newer and ten times faster), and decreasing by a factor of 10 (before inflation) every five years. The base figure of $25.00 includes the cost of designing boards, control software and manufacturing. The cost reported is the average of the cheapest solution and the second cheapest solution in order to discount economies of scale. . . . The results represent a guess with an error of perhaps 50%.

Table 7.3 gives different cost estimates based on the year the machine could be built, the key length, and the time required to recover the key. (The original tables also included estimates for inflation, which I omitted.) The 1990 estimates look at the average cost of a machine using chips capable of one million encryptions per second and a machine using chips capable of two million encryptions per second; chips capable of four million encryptions per second, while possible, are assumed to be too expensive to consider using. The 1995 estimates average two-million-encryptions-per-second machines and four-million-encryptions-per-second machines. The year 2000 estimates average four-million-encryptions-per-second machines and 32-million-encryptions-per-second machines.

For 56- and 64-bit keys, these numbers are within the realm of the military budgets of many industrialized nations. Breaking a 128-bit key is still far beyond the realm of possibility.

If attackers want to break a key badly enough, all they have to do is spend the requisite amount of money. Consequently, it seems prudent to try to estimate the minimum "value" of a key: how much value can be trusted to a single key before it makes economic sense to try to break? To give an extreme example, if a plaintext message worth $1.39 is encrypted with an algorithm, then it wouldn't make much

TABLE 7.3
Cost of a Brute-Force Cracking Machine

	MACHINE CAPABLE OF BREAKING A 56-BIT ALGORITHM WITHIN:			
YEAR	1 YEAR	1 MONTH	1 WEEK	1 DAY
1990	$130,000	$1.6M	$6.7M	$47M
1995	$64,000	$780,000	$3.3M	$23M
2000	$16,000	$190,000	$830,000	$5.8M
2005	$2000	$24,000	$100,000	$730,000
	MACHINE CAPABLE OF BREAKING A 64-BIT ALGORITHM WITHIN:			
YEAR	1 YEAR	1 MONTH	1 WEEK	1 DAY
1990	$33M	$400M	$1.7B	$12B
1995	$16M	$200M	$850M	$6.0B
2000	$4.1M	$50M	$210M	$1.5B
2005	$510,000	$6.2M	$27M	$190M
	MACHINE CAPABLE OF BREAKING A 128-BIT ALGORITHM WITHIN:			
YEAR	1 YEAR	1 MONTH	1 WEEK	1 DAY
1990	6.0×10^{26}	7.3×1027	3.2×10^{28}	2.2×10^{29}
1995	3.0×10^{26}	3.7×10^{27}	1.6×10^{28}	1.1×10^{29}
2000	7.5×10^{25}	9.2×10^{26}	3.9×10^{27}	2.8×10^{28}
2005	9.4×10^{24}	1.2×10^{26}	4.9×10^{26}	3.4×10^{27}

financial sense to set a $40-million cracker to the task of recovering the key. On the other hand, if the plaintext message is worth $100 million, then decrypting that single message would justify the cost of building the cracker alone, even today.

The question of when a key is worth attacking is also addressed in [367]. The authors assumed that attackers require a 25% annual return on their investment over a five-year period to cover the cost of financing construction of the machine and operating expenses. (25% is an arbitrary figure, but sufficient for estimating purposes.) Anything above that is profit; anything below that is loss.

Table 7.4 gives the attacker's cost per period: the cost required to recover a single key. If, over the course of five years, each key recovered is worth the cost to recover it, then the attacker breaks even. If each key is worth more, then the attacker makes a profit. This assumes that the cracking machine is running continuously—24 hours a day, 365 days a year. It does not assume that every key recovered is worth the minimum amount, only that on the average, every key is worth the minimum amount. A single high-value key can make the entire five-year operation profitable.

TABLE 7.4
Cost-per-Period of Breaking a 56-bit Key, Assuming Compounding-per-Period of 25% per Annum

"BREAK-EVEN" COST-PER-PERIOD TO BREAK A 56-BIT KEY IN:				
YEAR	1 YEAR	1 MONTH	1 WEEK	1 DAY
1990	$47,000	$46,000	$45,000	$45,000
1995	$24,000	$23,000	$22,000	$22,000
2000	$5,900	$5,700	$5,600	$5,600
2005	$740	$710	$700	$700
"BREAK-EVEN" COST-PER-PERIOD TO BREAK A 64-BIT KEY IN:				
YEAR	1 YEAR	1 MONTH	1 WEEK	1 DAY
1990	$12M	$12M	$12M	$11M
1995	$6.1M	$6.0M	$5.8M	$5.7M
2000	$1.5M	$1.5M	$1.4M	$1.4M
2005	$190,000	$180,000	$180,000	$190,000
"BREAK-EVEN" COST-PER-PERIOD TO BREAK A 128-BIT KEY IN:				
YEAR	1 YEAR	1 MONTH	1 WEEK	1 DAY
1990	$\$2.2\times10^{26}$	$\$2.2\times10^{26}$	$\$2.1\times10^{26}$	$\$2.1\times10^{26}$
1995	$\$1.1\times10^{26}$	$\$1.1\times10^{26}$	$\$1.1\times10^{26}$	$\$1.1\times10^{26}$
2000	$\$2.8\times10^{25}$	$\$2.7\times10^{25}$	$\$2.7\times10^{25}$	$\$2.6\times10^{25}$
2005	$\$3.5\times10^{24}$	$\$3.4\times10^{24}$	$\$3.3\times10^{24}$	$\$3.3\times10^{24}$

Software Crackers

Without special-purpose hardware and massively parallel machines, brute-force attacks are significantly harder. Software is so slow that economic brute-force attacks will be unfeasible for years to come, even for a 56-bit key. A Motorola 68040 microprocessor can perform an estimated 23,200 DES encryptions per second. If history is to be a guide, the speed increase has been sixteenfold over the past ten years. At this rate, within five years microprocessors will be able to run DES software at a rate of 64,000 encryptions per second. Within ten years that rate will increase to 256,000 encryptions per second. This is still a thousand times slower than today's best hardware implementations.

The real threat of a software-based, brute-force attack is not that it is certain, but that it is "free." There is no cost involved in setting up a microcomputer when it is idle to test possible keys. If it finds the correct key—great. If it doesn't, then nothing is lost. There is no cost involved in setting up an entire microcomputer

network to do that. A recent experiment with DES used the collective idle time of 40 workstations to test 2^{34} keys in a single day [367]. At this speed, it will take four million days to test all keys, but if enough people try attacks like this, then someone somewhere will get lucky.

> The crux of the software threat is sheer bad luck. Imagine a university computer network of 512 workstations, networked together. On some campuses this would be a medium-sized network. They could even be spread around the world, coordinating their activity through electronic mail. Assume each workstation is capable of running [the algorithm] at a rate of 15,000 encryptions per second. . . . Allowing for the overhead of testing and changing keys, this comes down to . . . 8,192 tests per second per machine. To exhaust [a 56-bit] keyspace with this setup would take 545 years (assuming the network was dedicated to the task twenty-four hours per day). Notice, however, that the same calculations give our hypothetical student hackers one chance in 200,000 of cracking a key in one day. Over a long weekend their odds increase to one chance in sixty-six thousand. The faster their hardware, or the more machines involved, the better their chance becomes. These are not good odds for earning a living from horse racing, but they're not the stuff of good press releases either. They are much better odds than the Government gives on its lotteries, for instance. "One-in-a-million"? "Couldn't happen again in a thousand years"? It is no longer possible to say such things honestly. Is this an acceptable ongoing risk? [367]

Using an algorithm with a 64-bit key instead of a 56-bit key makes this attack 256 times more difficult. With a 40-bit key, the picture is far more bleak. A network of 400 computers, each capable of performing 32,000 encryptions per second, can complete a brute-force attack against a 40-bit key in a single day. (In 1992, the RC2 and RC4 algorithms were approved for export with a 40-bit key—see Section 11.8.)

Using a 128-bit key makes this attack ridiculous even to contemplate. Industry estimates suggest that by 1996 there will be 200 million computers in use worldwide. This estimate includes everything from giant Cray mainframes to subnotebooks. If every one of those computers worked together on this brute-force attack, and each computer performed a million encryptions per second every second, it would still take a million times the age of the universe to recover the key.

Viruses

Probably the main difficulty with getting millions of computers to work on a brute-force attack is convincing the individual computer owners to let their computers participate. You could ask politely, but that's time-consuming. Even worse, they might say no. You could break into their machines at night, but that's even more time-consuming. Even worse, you might get arrested. Computer viruses could spread the cracking program more efficiently over as many computers as possible.

This is a particularly insidious idea, first presented in [875]. The attacker writes and lets loose a computer virus. This virus doesn't reformat the hard drive or delete files; all it does is work on a brute-force cryptanalysis problem whenever the

computer is idle. Various studies have shown that microcomputers are idle between 70% and 90% of the time, so the virus shouldn't have any trouble finding time to work on its task. If it is otherwise benign, it might even escape notice while it does its work.

Eventually, one machine will stumble on the correct key. At this point there are two ways of proceeding. First, the virus could spawn a different virus. It wouldn't do anything but reproduce and delete any copies of the cracking virus it finds but would contain the information about the correct key. This new virus would simply propagate through the computer world until it lands on the computer of the person who wrote the original virus.

A second, sneakier approach would be to display this message on the screen:

```
There is a serious bug in this computer.
Please call 1-800-123-4567 and read the
following 64-bit number to the operator:

xxxx xxxx xxxx xxxx

There is a $100 reward for the first
person to report this bug.
```

How efficient is this attack? Assume the typical infected computer can try a thousand keys per second. This is far less than the computer's maximum potential, because we have to assume it will be doing other things occasionally. Also assume that the typical virus infects ten million machines. This virus can break a 56-bit key in 83 days and a 64-bit key in 58 years. You might have to bribe the antiviral software makers, but that's your problem. Any increase in computer speeds or the virus infection rate would, of course, make this attack more efficient.

The Chinese Lottery

The Chinese Lottery is an eclectic, but possible, suggestion for a massively parallel DES cracker [718]. Imagine that a brute-force, million-test-per-second cracking chip was built into every radio and television sold. Each chip is programmed to test a different set of keys automatically upon receiving a plaintext/ciphertext pair over the airwaves. Then, every time the Chinese government wants to break a key, it broadcasts the data. All the radios and televisions in the country start chugging away. Eventually, the correct key will appear on someone's display, somewhere in the country. The Chinese pays a lottery prize to that person; this makes sure that the result is reported properly and also helps the sale of radios and televisions with the cracking chips.

If every man, woman, and child in China owns a radio or television, then the correct key to a 56-bit algorithm will appear in 64 seconds. If only one in ten Chinese owns a radio or television—closer to reality—the correct key will appear in eleven minutes. The correct key for a 64-bit algorithm will appear in 4.5 hours—45 hours if only one in ten owned a radio or television.

Some modifications are required to make this attack practical. First, it would be easier to have each chip try random keys instead of a unique set of keys. This

would make the attack about 37% slower—not much in light of the numbers we're working with. Also, the Chinese Communist party would have to mandate that every person listen to or watch a certain show at a certain time, just to make sure that all of the radios and televisions are operating when the plaintext/ciphertext pair is broadcast. Finally, everyone would have to be instructed to call a Central-Party-Whatever-It's-Called if their "Found the Key" light ever goes off, and then to read the string of numbers that appears on their screen. The Party could even make it a lottery; the person who finds the key wins a prize.

Table 7.5 shows the effectiveness of the Chinese Lottery for different countries and different key lengths. China is clearly in the best position to launch such an attack, but they would have to outfit every man, woman, and child with their own television or radio. The United States has fewer people but a lot more equipment per capita. The state of Wyoming could break a 56-bit key all by itself in less than a day.

Biotechnology

If biochips are possible, then it would be foolish not to use them as a distributed brute-force cryptanalysis tool. Consider a hypothetical animal, unfortunately called a "DESosaur" [718]. It consists of biological cells capable of testing possible keys. The plaintext/ciphertext pair is broadcast to the cells via some optical channel (these cells are transparent, you see). Solutions are carried to the DESosaur's speech organ via special cells that propagate through the animal's circulatory system.

The typical dinosaur has about 10^{14} cells (excluding bacteria). If each of them can perform a million encryptions per second (granted, this is a big if), breaking a 56-bit key would take seven ten-thousandths of a second. Breaking a 64-bit key would take less than two-tenths of a second. Breaking a 128-bit key would still take 10^{11} years, though.

Another biological approach is to use genetically engineered cryptanalytic algae that are capable of performing brute-force attacks against cryptographic

TABLE 7.5
Brute-Force Cracking Estimates for Chinese Lottery

LOCATION	POP. (APPROX.)	# OF TELEVISIONS & RADIOS	TIME TO BREAK 56-BIT ALGORITHM	TIME TO BREAK 64-BIT ALGORITHM
China	1,130,065,000	265,000,000	272 secs.	19 hrs.
U.S.	248,709,873	687,000,000	104 secs.	7.4 hrs.
Iraq	18,782,000	4,800,000	4.2 hrs.	44 days
Israel	4,371,000	2,620,000	7.6 hrs.	81 days
Wyoming	453,588	1,256,000	16 hrs.	170 days
Morehead, NC	6,046	16,700	50 days	35 yrs.

(Data on population and number of radios and televisions per capita taken from the *1992 World Almanac.*)

algorithms [718]. These organisms would make it possible to construct a distributed machine with more processors, because they could cover a larger area. The plaintext/ciphertext pair could be broadcast by satellite. If an organism found the result, it could induce the nearby cells to change color to communicate the solution back to the satellite.

Assume the typical algae cell is the size of a cube 10 microns on a side (this is probably a large estimate), then 10^{15} of them can fill a cubic meter. Pump them into the ocean and cover 200 square miles (518 square kilometers) of water to a meter deep (you figure out how to do it—I'm just the idea man), and you'd have 10^{23} (over a hundred billion gallons) of them floating in the ocean. (The *Exxon Valdez* spilled 10 million gallons of oil.) If each of them can try a million keys per second, they will recover the key for a 128-bit algorithm in just over 100 years. (The resulting algae bloom is your problem.) Breakthroughs in either algae processing speed, algae diameter, or even the size puddle one could spread across the ocean, would reduce these numbers significantly.

Don't even ask me about nanotechnology.

How Long Should a Key Be?

The answer depends on your application. How long does your data need to be secure? How much is your data worth? You can even specify the security requirement in this manner. For example:

> The key length must be such that there is a probability of no more than 1 in 2^{32} that an attacker with $100 million to spend could break the system, even assuming a growth rate of 30% per annum over the period.

Table 7.6, taken partially from [93], gives the estimated secrecy requirements for different kinds of information.

Future computing power is harder to estimate, but here is a reasonable rule of thumb: The efficiency of computing equipment divided by price increases by a factor of ten every five years [93]. Thus, in 50 years the fastest computers will be 10^{10} times faster than today! And these numbers only relate to general-purpose computers; who knows what kind of specialized cryptosystem-breaking equipment will be developed in the next 50 years?

Assuming that a cryptographic algorithm will be in use for 30 years, you can get some idea how secure it must be. An algorithm designed today probably will not see general use until 2000, and will still be used in 2030 to encrypt messages that must remain secret until 2080 or later.

7.2 KEY MANAGEMENT

Alice and Bob have a secure communications system. They play mental poker, simultaneously sign contracts, even securely evaluate circuits together. Their protocols are all secure. Their cryptographic algorithms are top-notch.

TABLE 7.6
Security Requirements for Different Information

TYPE OF TRAFFIC	LIFETIME	MIN. KEY LENGTH
Tactical military information	minutes/hours	56 bits
Product announcements, mergers, interest rates	days/weeks	56–64 bits
Trade secrets (e.g., recipe for Coca-Cola)	decades	64 bits
H-bomb secrets	> 40 years	128 bits
Identities of spies	> 50 years	128 bits
Personal affairs	> 50 years	128 bits
Diplomatic embarrassments	> 65 years	at least 128 bits
U.S. census data	100 years	at least 128 bits

Unfortunately, they buy their keys from Eve's "Keys-R-Us," whose slogan is "You can trust us: security is the middle name of our ex-mother-in-law's travel agent."

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

In the real world, key management is the hardest security problem. Designing secure cryptographic algorithms and protocols isn't easy, but making sure the keys stay secret is even harder. Cryptanalysts often attack both symmetric and public-key cryptosystems through their key management. Why bother going through all the trouble trying to break the cryptographic algorithm if the key can easily be recovered as a result of sloppy key management procedures? At the very least, Alice and Bob must protect the keys to the same degree as the data itself. The world's most secure algorithm won't help much if the users habitually choose their spouse's names for keys, or write their keys on little pieces of paper in their wallets. Further information on key management can be found in [269,53,714,685,452,214].

7.2.1 Generating Keys

The security of an algorithm rests in the key. If you're using a cryptographically weak process to generate your keys, then your whole system is weak. Eve doesn't have to attempt to cryptanalyze your cryptographic algorithm when she can cryptanalyze your key generation algorithm.

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

Reduced Keyspaces

DES has a 56-bit key. Implemented properly, any 56-bit string can be the key; there are 2^{56} (10^{16}) possible keys. One DES software encryption program only allows keys made up of lowercase letters and digits, allowing only 10^{12} possible keys. The poor key generation procedures in this program have made its DES ten thousand times easier to break than a proper implementation.

Table 7.7 gives the number of possible keys with various constraints on the input strings. Table 7.8 gives the time required for an exhaustive search through all of those keys, given a million attempts per second. Remember, the differences in the orders of magnitude mean that there is very little time differential between an exhaustive search for 8-byte keys and an exhaustive search of 4-, 5-, 6-, 7-, and 8-byte keys.

All of the specialized brute-force hardware and parallel implementations will work here. Testing a million keys per second (either with one machine or with multiple machines in parallel), it is feasible to crack lowercase-letter and lowercase-letter-and-number keys up to 8 bytes long, alphanumeric-character keys up to 7 bytes long, printable character and ASCII-character keys up to 6 bytes long, and 8-bit-ASCII-character keys up to 5 bytes long.

And remember, computers are continuously getting faster and cheaper. In other words, don't expect your keys to stand up against brute-force attacks for ten years.

Poor Key Choices

When people choose their own keys, they generally choose poor ones. They're far more likely to choose "Barney" than "*9(*hH*\A–". This is not always due to poor security practices; "Barney" is easier to remember than "*9(*hH*\A–". Any smart brute-force attack isn't going to try all possible keys in numerical order. It's going to try the easiest keys first: English words, names, trivial extensions to English words, etc. This is called a **dictionary attack,** because the attacker uses a dictionary of common keys. Daniel Klein was able to crack 25% of the passwords on the average computer using this system [482]:

1. Try using the user's name, initials, account name, and other relevant person information as a possible password. All in all, up to 130 different

TABLE 7.7
Number of Possible Keys of Various Keyspaces

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

TABLE 7.8
Exhaustive Search of Various Keyspaces (assume one million attempts per second)

	4-BYTES	5-BYTES	6-BYTES	7-BYTES	8-BYTES
Lowercase letters (26):	0.5 sec.	12 secs.	5 mins.	2.2 hrs.	2.4 days
Lowercase letters and digits (36):	1.7 sec.	1 min.	36 mins.	22 hrs.	33 days
Alphanumeric characters (62):	15.0 sec.	15 mins.	16 hrs.	41 days	6.9 yrs.
Printable characters (95):	1.4 mins.	2.1 hrs.	8.5 days	2.2 yrs.	210 yrs.
ASCII characters (128):	4.5 mins.	9.5 hrs.	51 days	18 yrs.	2,300 yrs.
8-bit ASCII characters (256):	1.2 hrs.	13 days	8.9 yrs.	2,300 yrs.	580,000 yrs.

passwords were tried based on this information. For an account name **klone** with a user named "Daniel V. Klein," some of the passwords that would be tried were: klone, klone0, klone1, klone123, dvk, dvkdvk, dklein, DKlein, leinad, nielk, dvklein, danielk, DvkkvD, DANIEL-KLEIN, (klone), KleinD, etc.

2. Try using words from various databases. These included lists of men's and women's names (some 16,000 in all); places (including permutations so that "spain," "spanish," and "spaniard" would all be considered); names of famous people; cartoons and cartoon characters; titles, characters, and locations from films and science fiction stories; mythical creatures (garnered from Bulfinch's mythology and dictionaries of mythical beasts); sports (including team names, nicknames, and specialized terms); numbers (both as numerals—"2001," and written out—"twelve"); strings of letters and numbers ("a," "aa," "aaa," "aaaa," etc.); Chinese syllables (from the Pinyin Romanization of Chinese, an international standard system of writing Chinese on an English keyboard); the King James Bible; biological terms; common and vulgar phrases (such as "fuckyou," "ibmsux," and "deadhead"); keyboard patterns (such as "qwerty," "asdf," and "zxcvbn"); abbreviations (such as "roygbiv"—the colors in the rainbow, and "ooottafagvah"—a mnemonic for remembering the 12 cranial nerves); machine names (acquired from */etc/hosts*); characters, plays, and locations from Shakespeare; common Yiddish words; the names of asteroids; and a collection of words from various technical papers I had previously published. All told, more than 60,000 separate words were considered per user (with any inter- and intra-dictionary duplicates being discarded).

3. Try various permutations on the words from step 2. This included making the first letter upper case or a control character, making the entire word upper case, reversing the word (with and without the aforementioned capitalization), changing the letter 'o' to the digit '0' (so that the word "scholar" would also be checked as "sch0lar"), changing the letter 'l' to

the digit '1' (so that the word "scholar" would also be checked as "scho1ar"), and performing similar manipulation to change the letter 'z' into the digit '2', and the letter 's' into the digit '5'. Another test was to make the word into a plural (irrespective of whether the word was actually a noun), with enough intelligence built in so that "dress" became "dresses," "house" became "houses," and "daisy" became "daisies." We did not consider pluralization rules exclusively, though, so that "datum" forgivably became "datums" (not "data"), while "sphynx" became "sphynxs" (and not "sphynges"). Similarly, the suffixes "-ed," "-er," and "-ing" were added to transform words like "phase" into "phased," "phaser," and "phasing." These 14 to 17 additional tests added another 1,000,000 words to the list of possible passwords that were tested for each user.

4. Try various capitalization permutations on the words from step 2 that were not considered in step 3. This included all single letter capitalization permutations (so that "michael" would also be checked as "mIchael," "miChael," "micHael," "michAel," etc.), double letter capitalization permutations ("MIchael," "MiChael," "MicHael," . . . , "mIChael," "mIcHael," etc.), triple letter permutations, etc. The single letter permutations added roughly another 400,000 words to be checked per user, while the double letter permutations added another 1,500,000 words. Three letter permutations would have added at least another 3,000,000 words per user had there been enough time to complete the tests. Tests of 4, 5, and 6 letter permutations were deemed to be impracticable without much more computational horsepower to carry them out.

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

6. Try word pairs. The magnitude of an exhaustive test of this nature is staggering. To simplify the test, only words of 3 or 4 characters in length from */usr/dict/words* were used. Even so, the number of word pairs is $O(10^7)$ (multiplied by 4096 possible salt values), and as of this writing, the test is only 10% complete.

A dictionary attack is much more powerful when it is used against a file of keys and not a single key. A single user may be smart enough to choose good keys. If

a thousand people each choose their own key as a password to a computer system, the odds are excellent that at least one person will choose a key in the attacker's dictionary.

Generating Good Keys

Good keys are random bit strings. If the key is 64 bits long, every possible 64-bit key must be equally likely. Generate the key bits from either a reliably random source (see Section 15.3), or a cryptographically secure pseudo-random bit generator (see Section 15.2). If these automatic processes are unavailable, flip a coin or roll a die.

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

Generating keys for public-key cryptography systems is much harder, because the keys must have certain mathematical properties (they may have to be prime, be a quadratic residue, etc.). Techniques for generating large random prime numbers are discussed in Section 9.5. The important thing to remember from a key management point of view is that the random seeds for those generators must be just that: random.

Generating a random key isn't always possible. Sometimes you need to remember your key (see how long it takes you to remember 25E8 56F2 E8BA C820). If you have to generate an easy-to-remember key, make it obscure. The ideal would be something easy to remember, but difficult to guess. Here are some suggestions:

- Word pairs separated by a punctuation character, for example "turtle*moose" or "zorch!splat"
- Strings of letters that are the initials to a longer phrase, for example "Mein Luftkissenfahrzeug ist voller Aale!" generates the key "MLivA!"

A better solution is a technique called **key crunching,** which converts easy-to-remember text strings into random keys. Use a one-way hash function to transform an arbitrary-length text string into a pseudo-random bit string.

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

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

might crunch into this 64-bit key:

E6C1 4398 5AE9 0A9B

The text string is often referred to as a **pass phrase.** If the phrase is long enough, the resulting key will be random. Exactly what "long enough" means is open to interpretation. For a 64-bit key, a pass phrase of about 10 to 15 normal English words should be sufficient. As a rule of thumb, figure that you need one letter of phrase for each bit of the key—64 letters for a 64-bit key. That's assuming only lowercase letters and spaces, without capitalization or punctuation. To increase the randomness of the key, the entire printable ASCII character set (upper- and lowercase characters, numbers, punctuation, and spaces) should be allowed in the pass phrase.

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

X9.17 Key Generation

The ANSI X9.17 standard specifies a method of key generation. This does not generate easy-to-remember keys; it is more suitable for generating session keys within a system. The cryptographic algorithm used to generate keys is DES, but it could just as easily be any algorithm.

Let $E_k(X)$ be DES encryption of X with key k. The key k is a key reserved for secret key generation. V_0 is a secret 64-bit seed. T is a timestamp. To generate the random key R_i, calculate:

$$R_i = E_k(E_k(T_i) \text{ XOR } V_i)$$

To generate V_{i+1}, calculate:

$$V_{i+1} = E_k(E_k(T_i) \text{ XOR } R_i)$$

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

7.2.2 Transferring Keys

Alice and Bob are going to use a conventional cryptographic algorithm to communicate securely; they need the same key. Alice generates a key using a random-key generator. Now she has to give it to Bob—securely. If Alice can meet Bob somewhere (a back alley, a windowless room, or the dark side of the moon), she can give him a copy of the key. Otherwise, they have a problem.

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

Public-key cryptography solves the problem nicely (see Section 3.1), but those techniques are not always available. Some systems use alternate channels known to be secure. Alice could send Bob the key with a trusted messenger. She could send it by certified mail or through an overnight delivery service. Alternatively, she could set up another communications channel with Bob and hope no one is eavesdropping on that one.

A better solution would be to split the key into several different parts and send each of those parts over a different channel. One part could be sent over the telephone, one by mail, one by overnight delivery service, one by carrier pigeon, etc. (see Figure 7.1). Since an adversary can collect all but one of the parts and still have no idea what the key is, this method will work in all but extreme cases. Section 3.5 discusses schemes for splitting a key into several parts.

However, if a system uses a new encryption key every day, it could quickly become cumbersome. One solution is for Alice to send Bob a month's worth of keys simultaneously. This key list is very valuable, and Alice and Bob must both securely store the list after receipt. Compromise the list, and the entire month of communications is at risk.

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

Key Distribution in Large Networks

Key-encryption keys shared by pairs of users works well in small networks, but can quickly get cumbersome if the networks become large. Since every pair of

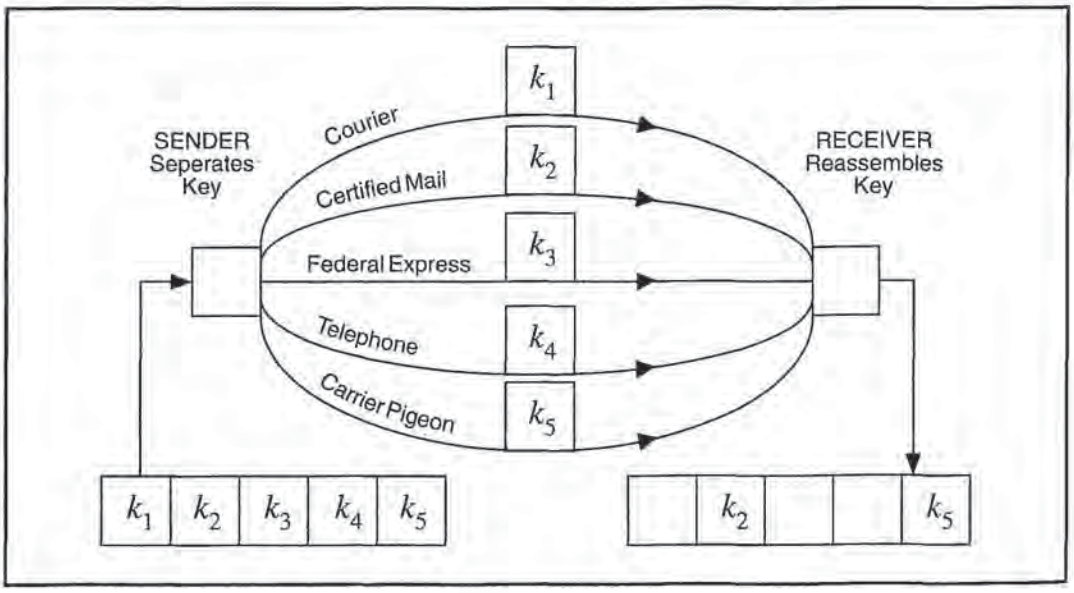


Figure 7.1
Key distribution in pieces.

users must exchange keys, the total number of key exchanges required in an n-person network is $n(n-1)/2$.

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

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

7.2.3 Verifying Keys

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

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

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

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

Bob could verify Alice's key over the telephone, where he can hear her voice. It's a public key; he can recite it over the telephone. If the key is too long, Alice and Bob can use a one-way hash function to verify the key. There are many possibilities.

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

Error Detection

Sometimes keys can get garbled in transmission. Since a garbled key can mean megabytes of undecryptable ciphertext, pains must be taken to minimize this

problem. All keys should be transmitted with some kind of error detection and correction bits. At the minimum, a checksum should be appended to the key. This way errors in transmission can be easily detected, and if required the key can be re-sent.

One of the most widely used methods is to encrypt a constant (either all 0's, or all 1's) with the key, and to send the first two to four bytes of the ciphertext along with the key. At the receiving end, do the same thing. If the encrypted constants match, then the key has been transmitted without error. The chance of an undetected error ranges from one in 2^{16} to one in 2^{24}.

7.2.4 Using Keys

Software encryption is scary. Gone are the days of microcomputers under the control of single programs. Now there's Macintosh System 7, Windows NT, and UNIX. There is no telling when the operating system will suspend the encryption application in progress, write everything to disk, and take care of some pressing task. When the operating system finally gets back to encrypting whatever is being encrypted, everything will look just fine. No one will ever realize that the operating system wrote the encryption application to disk, and that it wrote the key along with it. The key will sit on the disk, unencrypted, until the computer writes over that area of memory again. It could be minutes or it could be months. It could even be never; the key could still be sitting there when an adversary goes over the hard drive with a fine-tooth comb.

In a preemptive, multitasking environment, you can set your encryption operation to a high enough priority so it will not be interrupted. This would mitigate the risk. Even so, the whole thing is dicey at best.

Hardware implementations are easier. Many encryption devices are designed to erase the key if tampered with. For example, the IBM PS/2 encryption card has an epoxy unit containing the DES chip, battery, and memory. Of course, you have to trust the hardware manufacturer to implement the feature properly. Software implementations should try to do that as well, but nothing works perfectly.

7.2.5 Storing Keys

The easiest key storage problem is that of single users encrypting files for later use. Since they are the only persons involved, they are the only persons responsible for the key. Some systems take the easy approach: the key is stored in the user's brain and never on the system. The user is responsible for remembering the key and entering it every time a file is to be encrypted or decrypted.

An example of this system is IPS [501]. Users can either directly enter the 64-bit key, or users can enter the key as a long character string. The system then generates a 64-bit key from the character string using a key-crunching technique.

Another solution is to store the key in a ROM key or a magnetic stripe card [343,267]. Users could then enter their key into the system by inserting the key or card into a special reader attached to the encryption box or to the computer terminal.

This technique could be made even more secure by splitting the key into two halves, storing one half in the terminal and the other half in a ROM key. This way,

losing the ROM key does not compromise the key—change it and everything is back to normal. The same with the loss of the terminal. This way, compromise of either the ROM key or the system does not compromise the key. An adversary must have both parts to reconstruct the actual key.

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

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

7.2.6 Backup Keys

Alice is the chief financial officer at Secrets, Inc. Like any good corporate officer, she follows the company's security guidelines and encrypts all her data. Unfortunately, she ignores the company's street-crossing guidelines and gets hit by a truck. What does the company's president, Bob, do?

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

There are several ways Bob can avoid this. The simplest is to require all of his employees to write down their keys and give them to the company's security officer, who will lock them in a safe somewhere (or encrypt them all with a master key). Now, when Alice is bowled over on the interstate, Bob can ask his security officer for her key. Bob should make sure to have the combination to the safe himself as well, otherwise if the security officer is run over, Bob will be back where he started from.

The problem with this system is that Bob has to trust his security officer not to misuse everyone's keys. Even more significantly, all the employees have to trust the security officer not to misuse their keys. A far better solution is to use a secret-sharing protocol (see Section 3.6).

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

7.2.7 Compromised Keys

All of the protocols, techniques, and algorithms in this book are secure only if the key (the private key in the case of a public-key system) remains secret. If the key

is lost, stolen, printed in the newspaper, or otherwise compromised, then all security is lost.

There's not much to do if the key is lost. Ideally, users know when their key is compromised. If a KDC is managing the keys, users should notify it that the key has been compromised. If there is no KDC, then they should notify all correspondents who might receive messages from them. Someone should make public the fact that any message received after the key was lost is suspect, and that no one should send messages to the user with the key (or associated public key). The system should use some sort of timestamp, and then users can figure out which messages are legitimate and which are suspect.

If users don't know exactly when their key was compromised, things become even more difficult. They may want to back out of a contract because the person who stole the key signed it instead of them. If you always allow this, then you give anyone the ability to back out of a contract by claiming that the key was compromised before it was signed. It has to be a matter for an adjudicator to decide.

The compromise of a private key is generally more serious than the compromise of a secret key, because private keys are more permanent. Secret keys are changed regularly; public/private key pairs are not. If Bob gets access to Alice's private key, he can impersonate her on the network: reading encrypted mail, signing correspondence, entering into contracts, etc.

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

These procedures and tips aren't terribly optimal, but it's the best solution possible. The moral of the story is to protect keys, and protect private keys in the system above all else.

7.2.8 Lifetime of Keys

No encryption key should be used for an indefinite period. There are several reasons for this:

> The longer a key is used, the greater the chance that it is compromised. People write keys down; people lose them. Accidents happen. If you use the same key for a year, there's a far greater chance of compromise than if you use it for a day.
>
> The longer a key is used, the greater the loss if the key is compromised. If a key is used only to encrypt a single budgetary document on a file server, then the loss of the key means only the compromise of that document. If the same key is used to encrypt all the budgetary information on the file server, then its loss is much more devastating.
>
> The longer a key is used, the greater the temptation for someone to spend the effort necessary to break it—even if that effort is a brute-force attack. Breaking a key shared between two military units for a day would enable someone to read and fabricate messages between those units for that day.

Breaking a key shared by the entire military command structure for a year would enable that same person to read and fabricate messages throughout the world throughout the year. Which key do you think a nameless government agency, in our budget-conscious post–cold war world, would choose to attack?

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

For any cryptographic application, there must be a policy that determines the permitted lifetime of a key. Different keys may have different lifetimes. Communication session keys should have relatively short lifetimes, depending on the value of the data and the amount of data encrypted during a given period. The key for a gigabit-per-second communications link might have to be changed more often than the key for a 9600-baud modem link. Assuming there is an efficient method of transferring new keys using key-encryption keys, session keys should be changed at least daily.

On the other hand, key-encryption keys don't have to be replaced as frequently. They are used only occasionally (roughly once per day) for key exchange. There is little ciphertext for a cryptanalyst to work with. However, the potential loss if a key-encryption key is compromised is extreme: all communications encrypted with every key encrypted with the key-encryption key. In some applications, key-encryption keys are replaced just once a month or once a year.

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

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

7.2.9 Destroying Keys

Given that keys must be replaced regularly, old keys must be destroyed. Old keys are valuable, even if they are never to be used again. With them, an adversary can read old messages encrypted with that key [35].

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

If the key is in a hardware EEPROM, the key should be overwritten multiple times. If the key is in a hardware EPROM or PROM, the chip should be smashed into tiny bits and scattered to the four winds. If the key is stored on a computer disk, the actual bits of the storage should be overwritten multiple times (see Section 8.9).

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

7.3 PUBLIC-KEY KEY MANAGEMENT

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

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

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

Public-Key Certificates

A public-key certificate is someone's public key, signed by a trustworthy person. Certificates are used to thwart attempts to substitute one key for another. Bob's certificate, in the public-key database, contains a lot more than his public key. It contains information about Bob—his name, address, etc.—and it is signed by

someone Alice trusts: Trent (usually known as a **certifying authority,** or CA). By signing both the key and the information about Bob, Trent certifies that the information about Bob is correct, and that the public key belongs to Bob. Alice checks the signature, and then she can use the public key, secure in the knowledge that it is Bob's and no one else's. Certificates play an important role in a number of public-key protocols: PEM [476] (see Section 17.7) and CCITT X.509 [188] (see Section 17.6).

Distributed Key Management

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

Distributed key management, used in the public-domain program Pretty Good Privacy [908] (see Section 17.9), solves this problem with **introducers.** Introducers are other users of the system who sign their friends' public keys. For example, when Bob generates his public key, he gives copies to his friends Carol and Dave. They know Bob, so they each sign Bob's key and give Bob a copy of the signature. Now, when Bob presents his key to a stranger, Alice, he presents it with the signatures of these two introducers. If Alice also knows and trusts Carol, she has reason to believe that Bob's key is valid. If she knows and trusts Carol and Dave a little, she has reason to believe that Bob's key is valid. If she doesn't know either Carol or Dave, she has no reason to trust Bob's key.

Over time, Bob will collect many more introducers. If Alice and Bob travel in similar circles, the odds are good that Alice will know one of Bob's introducers.

To prevent against Mallet's substituting one key for another, introducers must be sure that Bob's key belongs to Bob before they sign it. Perhaps introducers should require the key to be given face-to-face or verified over the telephone.

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

Public-Key Algorithms

12.1 BACKGROUND

The concept of public-key cryptography was invented by Whitfield Diffie and Martin Hellman, and independently by Ralph Merkle. This contribution to the field was the notion that keys could come in pairs—an encryption key and a decryption key—and that it could be unfeasible to generate one key from the other. Diffie and Hellman first presented this concept at the 1976 National Computer Conference [298]; a few months later, their seminal paper, "New Directions in Cryptography," appeared in *IEEE Transactions on Information Theory* [299]. (Due to the glacial publishing speed of *Communications of the ACM,* Merkle's first contribution to the field didn't appear until 1978 [586].)

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 [297]:

> 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 [624] and technical

papers [483,645] 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* [824], "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.

Since 1976, numerous public-key cryptography algorithms have been proposed. Many of these are insecure. Of the ones that are still considered secure, many are impractical. Either they have an impractically large key, or the ciphertext is much larger than the plaintext.

Only a few algorithms are both secure and practical. These algorithms are generally based on one of the hard problems discussed in Section 9.2. While it is theoretically possible that someone may come up with a fast solution tomorrow, not many experts think it is likely.

Of these secure and practical public-key algorithms, some are only suitable for key distribution. Others are suitable for encryption (and by extension for key distribution). Others are only suitable for digital signatures. Only two algorithms are suitable for both encryption and digital signatures: RSA and ElGamal. All of these algorithms are slow. The encryption and decryption rate is much slower than with symmetric algorithms; usually it's too slow to support bulk data encryption.

Security of Public-Key Algorithms

Since cryptanalysts have access to the public key, they can always choose any message to encrypt. This means that cryptanalysts, given $C = E_K(P)$, can guess the value of P and easily check their guess. This is a serious problem if the number of possible plaintext messages is small enough to allow exhaustive search, but can be solved by padding messages with a string of random bits. This means that identical plaintext messages will encrypt to different ciphertext messages. (For more about this concept, see Section 16.16.)

Public-key algorithms are designed to resist chosen-plaintext attacks; their security is based both on the difficulty of deducing the secret key from the public key and the difficulty of deducing the plaintext from the ciphertext. However, most public-key algorithms are particularly susceptible to what is called a **chosen-ciphertext attack:**

A chosen-ciphertext attack is when cryptanalysts choose messages and have access to the decryption of those messages with the private key. Their job is to deduce the key or an algorithm to decrypt any new messages encrypted with the same key.

Given: $C_1, P_1 = D_K(C_1), C_2, P_2 = D_K(C_2), \ldots C_i, P_i = D_K(C_i)$, where cryptanalysts choose $C_1, C_2, \ldots C_i$

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

It is important to understand the significance of this attack. Even if we have a public-key cryptosystem that is provably secure and use a protocol that looks safe,

cryptanalysts may still recover plaintext messages. The inherent problem is a direct result of the most useful characteristic of public-key cryptography: everyone can use the public key.

Consequently, it is important to look at the system as a whole and not just at the individual parts. Good public-key protocols are designed so that the various parties don't decrypt arbitrary messages generated by other parties—the proof-of-identity protocols are a good example (see Section 5.4).

12.2 DIFFIE-HELLMAN

Diffie-Hellman is the first public-key algorithm invented [299]. It gets its security from the difficulty of calculating discrete logarithms in a finite field, as compared with the ease of calculating exponentiation in the same field. Diffie-Hellman can be used for key distribution, but it cannot be used to encrypt and decrypt messages. Alice and Bob can use this algorithm to generate a secret key.

The math is simple. First, Alice and Bob agree on two large integers, n and g, such that g is less than n but greater than 1. These two integers don't have to be secret; Alice and Bob can agree to them over some insecure channel. They can even be common among a group of users. It doesn't matter.

Then, the protocol goes as follows:

(1) Alice chooses a random large integer x and computes

$$X = g^x \bmod n$$

(2) Bob chooses a random large integer y and computes

$$Y = g^y \bmod n$$

(3) Alice sends X to Bob, and Bob sends Y to Alice. (Note that Alice keeps x secret, and Bob keeps y secret.)

(4) Alice computes $k = Y^x \bmod n$.

(5) Bob computes $k' = X^y \bmod n$.

Both k and k' are equal to $g^{xy} \bmod n$. No one listening on the channel can compute that value; they only know n, g, X, and Y. Unless they can compute the discrete logarithm and recover x or y, they do not solve the problem. So, k is the secret key that both Alice and Bob computed independently.

The choice of g and n can have a substantial impact on the security of this system. The modulus n should be a prime; more importantly $(n-1)/2$ should also be a prime [703]. And g should be a primitive root mod n. And most important, n should be large: at least 512 bits long. Using 1028 bits would be better.

Diffie-Hellman Extended

This algorithm also works in finite fields [703]. Shmuley and Kevin McCurley studied a variant of the algorithm where the modulus is a composite number [812,578]. V. S. Miller and Neal Koblitz extended this algorithm to elliptic curves

[605,489]. Taher ElGamal used the basic idea to develop an encryption and digital signature algorithm (see Section 13.3).

This algorithm also works in the Galois field GF(2^k) [812,578]. Some implementations take this approach [504,896,897], because the computation is much quicker. Similarly, cryptanalytic computation is also much quicker, so it is important to choose carefully a field large enough to ensure security.

Diffie-Hellman with Three or More Parties

The Diffie-Hellman key-exchange protocol can easily be extended to work with three or more people. In this example, Alice, Bob, and Carol together generate a secret key:

(1) Alice chooses a random large integer x and computes

$$X = g^x \bmod n$$

(2) Bob chooses a random large integer y and computes

$$Y = g^y \bmod n$$

(3) Carol chooses a random large integer z and computes

$$Z = g^z \bmod n$$

(4) Alice sends X to Bob, Bob sends Y to Carol, and Carol sends Z to Alice.

(5) Alice computes $Z' = Z^x \bmod n$.

(6) Bob computes $X' = X^y \bmod n$.

(7) Carol computes $Y' = Y^z \bmod n$.

(8) Alice sends Z' to Bob, Bob sends X' to Carol, and Carol sends Y' to Alice.

(9) Alice computes $k = Y'^x \bmod n$.

(10) Bob computes $k = Z'^y \bmod n$.

(11) Carol computes $k = X'^y \bmod n$.

The secret key, k, is equal to $g^{xyz} \bmod n$, and no one else listening in on the communications can compute that value. The protocol can be easily extended to four or more people; just add more people and more rounds of computation.

Patents

The Diffie-Hellman key-exchange algorithm is patented in the United States [426] and Canada [427]. A group called Public Key Partners (PKP) licenses the patent, along with other public-key cryptography patents. Anyone interested in obtaining a license should contact:

Robert B. Fougner
Director of Licensing
Public Key Partners

130 B Kifer Court
Sunnyvale, CA 94086
Tel: (408) 735-6779

The U.S. patent will expire on April 29, 1997.

12.3 KNAPSACK ALGORITHMS

The first algorithm for generalized public-key encryption was developed by Ralph Merkle and Martin Hellman [420,593]. It could only be used for encryption, although Shamir later adapted the system for digital signatures [787]. Knapsack algorithms get their security from the knapsack problem, an NP-complete problem. Although this algorithm was later found to be insecure, it is worth examining because it demonstrates how an NP-complete problem can be used for public-key cryptography.

The knapsack problem is a simple one. Given a pile of items, each with different weights, is it possible to put some of those items into a knapsack so that the knapsack weighs a given amount?

For example, the items might have weights of 1, 5, 6, 11, 14, and 20. It is possible to pack a knapsack that weighs 22; use weights 5, 6, and 11. It is impossible to pack a knapsack that weighs 24. In general, the time required to solve this problem seems to grow exponentially with the number of items in the pile.

In general, the problem can be defined as follows: Given a set of values, M_1, M_2, . . . M_n, and a sum, S, compute the values of b such that:

$$S = b_1M_1 + b_2M_2 + \ldots + b_nM_n$$

The values of b can be either 0 or 1. A 1 indicates that the item is in the knapsack; a 0 indicates that it isn't.

The idea behind the Merkle-Hellman knapsack algorithm is to encode a message as a solution to a series of knapsack problems. A block of plaintext equal in length to the number of items in the pile would select the items (bits of the plaintext correspond to the b values: a 1 means the item was present, and a 0 means the item was absent), and the ciphertext would be the resulting sum. Figure 12.1 shows a plaintext encrypted with a sample knapsack problem.

The trick is that there are actually two different knapsack problems, one solvable in linear time and the other solvable only in exponential time. The easy knapsack can be modified to create the hard knapsack. The public key is the hard knapsack, which can easily be used to encrypt messages but cannot be used to decrypt

Plaintext:	1 1 1 0 0 1	0 1 0 1 1 0	0 0 0 0 0 0	0 1 1 0 0 0
Knapsack:	1 5 6 11 14 20	1 5 6 11 14 20	1 5 6 11 14 20	1 5 6 11 14 20
Ciphertext:	1+5+6+20=	5+11+14=	0=	5+6=
	32	30	0	11

Figure 12.1
Encryption with knapsacks.

messages. The private key is the easy knapsack, which gives an easy way to decrypt messages. People who don't know the private key are forced to try to solve the hard knapsack problem.

Superincreasing Knapsacks

What is the easy knapsack problem? If the list of weights is a **superincreasing sequence,** then the resulting knapsack problem is easy to solve. A superincreasing sequence is a sequence in which every term is greater than the sum of all the previous terms. For example, {1,3,6,13,27,52} is a superincreasing sequence, but {1,3,4,9,15,25} is not.

The solution of a **superincreasing knapsack** is easy to find. Take the total weight and compare it with the largest number in the sequence. If the total weight is less than the number, then it is not in the knapsack. If the total weight is greater than the number, then it is in the knapsack. Reduce the weight of the knapsack by the value and move to the next largest number in the sequence. Repeat until finished. If the total weight has been brought to zero, then there is a solution. If the total weight has not, there isn't.

For example, consider a total knapsack weight of 70 and a sequence of weights of {2,3,6,13,27,52}. The largest weight, 52, is less than 70, so 52 is in the knapsack. Subtracting 52 from 70 leaves 18. The next weight, 27, is greater than 18, so 27 is not in the knapsack. The next weight, 13, is less than 18, so 13 is in the knapsack. Subtracting 13 from 18 leaves 5. The next weight, 6, is greater than 5, so 6 is not in the knapsack. Continuing this process will show that both 2 and 3 are in the knapsack. Were this a Merkle-Hellman knapsack encryption block, the plaintext that resulted from a ciphertext value of 70 would be 110101.

With non-superincreasing knapsacks, there is no quick algorithm. The only way to determine which items are in the knapsack is to methodically test possible solutions until you stumble on the correct one. The fastest algorithms, taking into account the necessary heuristics, grow exponentially with the number of possible weights in the knapsack. Add one item to the sequence of weights, and it takes twice as long to find the solution. This is much more difficult than a superincreasing knapsack, when if you add one more weight to the sequence it simply takes another operation to find the solution.

The Merkle-Hellman algorithm is based on this property. The private key is a sequence of weights for a superincreasing knapsack problem. The public key is a sequence of weights for a normal knapsack problem with the same solution. Merkle and Hellman developed a technique for converting a superincreasing knapsack problem into a normal knapsack problem. They did this using modular arithmetic.

Creating the Public Key from the Private Key

Without going into the number theory, this is how the algorithm works: To get a normal knapsack sequence, take a superincreasing knapsack sequence, for example {2,3,6,13,27,52}, and multiply all of the values by a number n, mod m. The modulus should be a number greater than the largest number in the sequence, for example, 56. The multiplier should have no factors in common with any of the

numbers in the sequence, for example, 31. The normal knapsack sequence would then be:

$$2 \times 31 \bmod 56 = 6$$
$$3 \times 31 \bmod 56 = 37$$
$$6 \times 31 \bmod 56 = 18$$
$$13 \times 31 \bmod 56 = 11$$
$$27 \times 31 \bmod 56 = 53$$
$$52 \times 31 \bmod 56 = 44$$

The hard knapsack would then be {6,37,18,11,53,44}.

The superincreasing knapsack sequence is the private key. The normal knapsack sequence is the public key.

Encryption

To encrypt a binary message, first break it up into blocks equal to the number of items in the knapsack sequence. Then, allowing a 1 to indicate the item is present and a 0 to indicate that the item is absent, compute the total weights of the knapsacks.

For example, if the message were 011000110101101110, encryption using the above knapsack would proceed like this:

message = 011000 110101 101110

011000 corresponds to 37+18 = 55
110101 corresponds to 6+37+11+44 = 98
101110 corresponds to 6+18+11+53 = 88

The ciphertext would be

55,98,88

Decryption

A legitimate recipient of this message knows the original superincreasing knapsack as well as the values of n and m used to transform it into a normal knapsack. To decrypt the message, the recipient must first determine n^{-1} such that $n \times n^{-1} = 1 \bmod m$. Multiply each of the ciphertext values by $n^{-1} \bmod m$ to get the plaintext values.

In our example, the superincreasing knapsack is {2,3,6,13,27,52}, m is equal to 56, and n is equal to 31. The ciphertext message is 55,98,88. In this case n^{-1} is equal to 47, so the ciphertext values must be multiplied by 47 mod 56.

$$55 \times 47 \bmod 56 = 48 = 011000$$
$$98 \times 47 \bmod 56 = 53 = 110101$$

$$88 \times 47 \bmod 56 = 46 = 101110$$

The recovered plaintext is 011000 110101 101110.

Practical Implementations

With a knapsack sequence of only six items long, it's not hard to solve the problem even if it isn't superincreasing. Real knapsacks contain about 200 items. The value for each term in the superincreasing knapsack is somewhere between 200 and 400 bits long. The modulus is chosen to be somewhere between 100 to 200 bits long. Real implementations of the algorithm use random sequence generators to produce these values.

With knapsacks like that, it's futile to try to solve them by brute force. If a computer could try a million possibilities per second, trying all possible knapsack values would take over 10^{46} years. Even a million machines working in parallel wouldn't solve this problem before the sun went nova.

Security of Knapsacks

It wasn't a million machines that broke the knapsack cryptosystem, but a pair of cryptographers. First, Herlestam noticed that often a single bit of the plaintext could be recovered [432]. Then, Shamir showed that the algorithm can be broken in certain circumstances [789,790]. There were other results—[800,21,442,313, 291]—but no one could break the general Merkle-Hellman system. Finally, Shamir and Zippel [792,793,795] found flaws in the transformation that allowed them to reconstruct the superincreasing knapsack from the normal knapsack. The exact arguments are beyond the scope of this book, but a nice summary of them can be found in [687,696]. There is also a nice story about the conference, where the results were presented in [295,297] (the attack was demonstrated on stage using an Apple II computer).

Knapsack Variants

Since the original Merkle-Hellman scheme was broken, many other knapsack systems have been proposed: multiple iterated knapsacks, Graham-Shamir knapsacks, etc. These have all been analyzed and broken, generally using the same cryptographic techniques [162,156,170,516,10,514,515,517,221,157,164,158]. (Many more proposed knapsack algorithms litter the cryptologic highway; a good overview of these systems and their cryptanalysis can be found in [168,282,160,169].)

Other cryptographic systems have been proposed that use ideas similar to those used in knapsack cryptosystems, but these too have been broken. The Lu-Lee cryptosystem [558,8] was broken in [13,372,494]. Attacks on the Niederreiter cryptosystem [648] can be found in [168,169], as can attacks against the Goodman-McAuley cryptosystem [386,387]. The Pieprzyk cryptosystem [697] can be broken by similar attacks. The Niemi cryptosystem [649], based on modular knapsacks, was broken in [209,458].

While there is still a variation of the knapsack algorithm that is currently still secure—the Chor-Rivest knapsack [213]—the amount of computation required

makes it far less useful than the other algorithms discussed here. And even worse, considering the ease with which all the other variations fell, it doesn't seem prudent to trust them.

Patents

The original Merkle-Hellman algorithm is patented in the United States [428] and worldwide (see Table 12.1). PKP licenses the patent, along with other public-key cryptography patents. Anyone interested in obtaining a license should contact:

Robert B. Fougner
Director of Licensing
Public Key Partners
130 B Kifer Court
Sunnyvale, CA 94086
Tel: (408) 735-6779

The U.S. patent will expire on August 19, 1997.

TABLE 12.1
Foreign Merkle-Hellman Knapsack Patents

COUNTRY	NUMBER	DATE
Belgium	871039	5 Apr 1979
Netherlands	7810063	10 Apr 1979
Great Britain	2006580	2 May 1979
Germany	2843583	10 May 1979
Sweden	7810478	14 May 1979
France	2405532	8 Jun 1979
Germany	2843583	3 Jun 1982
Germany	2857905	15 Jul 1982
Canada	1128159	20 Jul 1982
Great Britain	2006580	18 Aug 1982
Switzerland	63416114	14 Jan 1983
Italy	1099780	28 Sep 1985

12.4 RSA

Soon after Merkle's knapsack algorithm came the first full-fledged public-key algorithm, one that works for encryption as well as for digital signatures. Of all the public-key algorithms proposed over the years, it is by far the easiest to understand and implement. (Martin Gardner published an early description of the algorithm in his "Mathematical Games" column in *Scientific American* [365].) It

is also the most popular. Named after the three inventors, Ron Rivest, Adi Shamir, and Leonard Adleman, who first introduced the algorithm in 1978 [749,750], it has since withstood years of extensive cryptanalysis. Although the cryptanalysis neither proved nor disproved RSA's security, it does suggest a confidence level in the theoretical underpinnings of the algorithm.

RSA gets its security from the difficulty of factoring large numbers. The public and private keys are functions of a pair of large (100 to 200 digits or even larger) prime numbers. Recovering the plaintext from one of the keys and the ciphertext is conjectured to be equivalent to factoring the product of the two primes.

To generate the two keys, choose two large prime numbers, p and q. Compute the product:

$$n = p \times q$$

Then randomly choose the encryption key, e, such that e and $(p-1) \times (q-1)$ are relatively prime. Finally, use Euclid's algorithm to compute the decryption key, d, such that

$$e \times d = 1 \pmod{(p-1) \times (q-1)}$$

In other words,

$$d = e^{-1} \pmod{(p-1) \times (q-1)}$$

Note that d and n are also relatively prime. The numbers e and n are the public key; the number d is the private key. The two primes, p and q, are no longer needed. They should be discarded, but never revealed.

To encrypt a message m, first divide it into numerical blocks such that each block has a unique representation modulo n (with binary data, choose the largest power of 2 less than n). That is, if both p and q are 100-digit primes, then n will have just under 200 digits, and each message block, m_i, should be just under 200 digits long. The encrypted message, c, will be made up of similarly sized message blocks, c_i, of about the same length. The encryption formula is simply:

$$c_i = m_i^e \pmod{n}$$

To decrypt a message, take each encrypted block c_i and compute:

$$m_i = c_i^d \pmod{n}$$

Since:

$$c_i^d = (m_i^e)^d = m_i^{ed} = m_i^{k(p-1)(q-1)+1} = m_i \times m_i^{k(p-1)(q-1)} = m_i \times 1 = m_i,$$
all mod n

the formula recovers the message. This is summarized in Table 12.2.

TABLE 12.2
RSA Encryption

PUBLIC KEY:

n product of two primes, p and q (p and q must remain secret)

e relatively prime to $(p-1) \times (q-1)$

PRIVATE KEY:

d $e^{-1} \pmod{(p-1) \times (q-1)}$

ENCRYPTING:

$c = m^e \pmod{n}$

DECRYPTING:

$m = c^d \pmod{n}$

The message could just as easily have been encrypted with d and decrypted with e; the choice is arbitrary. I am not including the number theory that proves why this works; most current texts on cryptography cover the theory in detail.

A short example will probably go a long way to making this clearer. If $p = 47$ and $q = 71$, then

$$n = p \times q = 3337$$

The encryption key e must have no factors in common with:

$$(p-1) \times (q-1) = 46 \times 70 = 3220$$

Choose e (at random) to be 79. In that case:

$$d = 79^{-1} \pmod{3220} = 1019$$

This number was calculated using the extended Euclidean algorithm (see Section 9.3). Publish e and n, and keep d secret. Discard p and q.

To encrypt the message

$$m = 6882326879666683$$

first break it into small blocks. Three-digit blocks work nicely in this case. The message will be encrypted in six blocks, m_i, in which:

$$m_1 = 688$$
$$m_2 = 232$$
$$m_3 = 687$$
$$m_4 = 966$$
$$m_5 = 668$$
$$m_6 = 3$$

The first block is encrypted as:

$$688^{79} \pmod{3337} = 1570 = c_1$$

Performing the same operation on the subsequent blocks generates an encrypted message:

$$c = 1570\ 2756\ 2714\ 2276\ 2423\ 158$$

Decrypting the message requires performing the same exponentiation using the decryption key of 1019. So:

$$1570^{1019} \pmod{3337} = 688 = m_1$$

The rest of the message can be recovered in this manner.

Security of RSA

The security of RSA depends wholly on the problem of factoring large numbers. Technically that's not true. It is conjectured that the security of RSA depends on the problem of factoring large numbers. It has never been mathematically proven that you need to factor n to calculate m from c and e. It is conceivable that an entirely different way to cryptanalyze RSA might be discovered. However, if this new way allows the cryptanalyst to deduce d, it could also be used as a new way to factor large numbers. I wouldn't worry about it too much.

For the ultra-skeptical, there are RSA variants that have been proven to be as difficult as factoring (see Section 12.6). Also look at [20], which shows that recovering even a single bit of information from an RSA-encrypted ciphertext is as hard as decrypting the entire message.

Nevertheless, factoring n is the most obvious means of attack. Adversaries will have the public key, e, and the modulus, n. To find the decryption key, d, they have to factor n. Section 9.4 discusses the current state of factoring technology. Currently, 110-digit numbers are regularly being factored, and 120-digit numbers will be able to be factored soon. So, n must be larger than that. Some hardware implementations of RSA use 512-bit (154-digit) values for n. I've seen 664-bit (200-digit) values for n in some software implementations. The most paranoid use RSA with 1024-bit (308-digit) n's. Also read about choosing good primes (Section 9.5).

Assuming the complexity of factoring given in Section 9.4, the factoring of a 664-bit number takes 10^{23} steps. Assuming one computer can perform a million steps per second, and that a million-computer network works on the task, it will take almost 4000 years to factor the number. If n is a 1024-bit number, that same array of computers will take 10^{10} years to factor the number.

At a European Institute for System Security workshop, the participants agreed that a 1024-bit modulus should be sufficient for long-term secrets for the next ten years [93]. However, some speakers warned: "Although the participants of this

workshop feel best qualified in their respective areas, this statement [with respect to lasting security] should be taken with caution." Caveat emptor.

RSA in Hardware

Much has been written on the subject of hardware implementations of RSA [739,831,819,741,840,495,682,46,786,785,753,222,801]. A good survey article is [161]. Many different chips that perform RSA encryption have been built [737,155,607,742,495,37,439,364,716,861,307,683].

A partial list of currently available RSA chips, from [93,161], is given in Table 12.3.

TABLE 12.3
Existing RSA Chips

COMPANY	CLOCK SPEED	BAUD RATE PER 512 BITS	CLOCK CYCLES PER 512 BIT ENCRYPTION	TECHNOLOGY	BITS PER CHIP	NUMBER OF TRANSISTORS
Alpha Techn.	25MHz	13 K	.98 M	2 micron	1024	180,000
AT&T	15MHz	19 K	.4 M	1.5 micron	298	100,000
British Telecom.	10MHz	5.1 K	1 M	2.5 micron	256	—
Business Sim. Ltd.	5MHz	3.8 K	.67 M	Gate Array	32	—
Calmos Syst. Inc.	20MHz	28 K	.36 M	2 micron	593	95,000
CNET	25MHz	5.3 K	2.3 M	1 micron	1024	100,000
Cryptech	14MHz	17 K	.4 M	Gate Array	120	33,000
Cylink	16MHz	6.8 K	1.2 M	1.5 micron	1024	150,000
Pijnenburg	25MHz	50 K	.256 M	1 micron	1024	400,000
Plessy Crypto.	—	10.2 K	—	—	512	—
Sandia	8MHz	10 K	.4 M	2 micron	272	86,000

Speed of RSA

At its fastest, RSA is about 1000 times slower than DES. The fastest VLSI hardware implementation for RSA with a 512-bit moduli has a throughput of 64 kbps [161]. There are also chips that perform 1024-bit RSA encryption. Currently chips are being planned that will approach 1 mbps using a 512-bit modulus; they will probably be available in 1994. Some manufacturers have implemented RSA in smart cards; these implementations are slower.

In software, DES is about 100 times faster than RSA. These numbers may change slightly as technology changes, but RSA will never approach the speed of symmetric algorithms. This is why most practical systems use RSA only to exchange DES keys, and then use DES to encrypt everything else.

Chosen Ciphertext Attack Against RSA

Scenario 1: Eve, listening in on Alice's communications, manages to collect a ciphertext message, c, encrypted with RSA in her public key. Eve wants to be able to read the message. Mathematically, she wants p, in which:

$$p = c^d$$

To recover p, she first chooses a random number, r, such that r is less than n. She gets Alice's public key, e. Then she computes:

$$x = r^e \bmod n$$
$$y = x \times c \bmod n$$
$$t = r^{-1} \bmod n$$

If $x = r^e \bmod n$, then $r = x^d \bmod n$. Therefore, $t = x^{-d} \bmod n$.

Now, Eve gets Alice to sign y with her private key, thereby decrypting y. (Alice has to sign the message, not the hash of the message.) Remember, Alice has never seen y before. Alice sends Eve:

$$u = y^d \bmod n$$

Now, Eve computes:

$$t \times u \bmod n = x^{-d} \times y^d \bmod n = x^{-d} \times x^d \times c^d \bmod n = c^d \bmod n = p$$

Eve now has p.

Scenario 2: Trent is a computer notary public. If Alice wants a document notarized, she sends it to Trent. Trent signs it with an RSA digital signature and sends it back. (No one-way hash functions are used here; Trent encrypts the entire message with his private key.)

Mallet wants Trent to sign a message he otherwise wouldn't. Maybe it has a phony timestamp; maybe it purports to be from another person. Whatever the reason, Trent would never sign it if he had a choice. Let's call this message N.

First, Mallet chooses an arbitrary value X and computes $Y = X^e$. He can easily get e; it's Trent's public key and must be public to verify his signatures. Then he computes $M = YN$, and sends M to Trent to sign. Trent returns $M^d \bmod n$. Now Mallet calculates $(M^d \bmod n)X^{-1}$, which equals $N^d \bmod n$ and is the signature of N.

Actually, there are several methods that Mallet can use to accomplish these same things [240,270,289]. The weakness they all exploit is the fact that exponentiation preserves the multiplicative structure of the input. That is:

$$(XM)^d = X^d M^d \bmod n$$

Scenario 3: Eve wants to Alice to sign M_3. She generates two messages, M_1 and M_2, such that:

$$M_3 \equiv M_1 \times M_2 \pmod{n}$$

If she gets Alice to sign M_1 and M_2, she can calculate M_3:

$${M_3}^d \pmod{n} = {M_1}^d \pmod{n} \times {M_2}^d \pmod{n}$$

Moral: Never use RSA to sign a random document presented to you by a stranger. Always use a one-way hash function first or a different signature algorithm altogether.

Common Modulus Attack on RSA

Let's make life easier, and give everyone the same n. They can have different values for e and d, but the same n. Unfortunately, this doesn't work. The most obvious problem is that if the same message is ever encrypted with two different keys (both having the same modulus), and those two keys are relatively prime (which they generally would be), then the plaintext can be recovered without either of the decryption keys [820].

Let P be the plaintext message. The two encryption keys are e_1 and e_2. The common modulus is n. The two ciphertext messages are:

$$C_1 = P^{e_1} \bmod n$$
$$C_2 = P^{e_2} \bmod n$$

The cryptanalysts know n, e_1, e_2, C_1, and C_2. Here's how they recover P.

Since e_1 and e_2 are relatively prime, the Euclidean algorithm can find r and s, such that:

$$r \times e_1 + s \times e_2 = 1$$

Assuming r is negative (either r or s has to be), then the Euclidean algorithm can be used again to calculate ${C_1}^{-1}$. Then:

$$({C_1}^{-1})^{-r} \times {C_2}^s = P \bmod n$$

There are two other, more subtle, attacks against this type of system. One attack uses a probabilistic method for factoring n. The other uses a deterministic algorithm for calculating someone's secret key without factoring the modulus. Both attacks are described in detail in [260]. Moral: Don't share n among a group of users.

Low Exponent Attack Against RSA

Another suggestion to "improve" RSA is to use low values for e, the public key. This makes encryption fast and easy to perform. Unfortunately, it is also insecure. Hastad demonstrated a successful attack against RSA with a low encryption key [417]. Another attack by Michael Wiener will recover e, when e is up to one quarter

the size of n [878]. A low decryption key, d, is just as serious a problem. Moral: Choose large values for e and d.

Lessons Learned

Moore lists several restrictions on the use of RSA, based on the success of these attacks [618,619]:

- Knowledge of one encryption/decryption pair of exponents for a given modulus enables an attacker to factor the modulus.
- Knowledge of one encryption/decryption pair of exponents for a given modulus enables an attacker to calculate other encryption/decryption pairs without having to factor n.
- A common modulus should not be used in a protocol using RSA in a communications network. (This should be obvious from the previous two points.)
- The exponents chosen in a protocol should not be small.

Remember, it is not enough to have a secure cryptographic algorithm. The entire cryptosystem must be secure, and the cryptographic protocol must be secure. If there is a failure in any of those three areas, the overall system is insecure.

Standards

RSA is a de facto standard in much of the world. The International Organization of Standards, ISO/IEC 9796, specifies RSA for digital signatures [446]. The French banking community has also standardized on RSA [310]. The United States currently has no standard for public-key encryption but will probably standardize on the Digital Signature Standard (see Section 13.5) for digital signatures.

Patents

The RSA algorithm is patented in the United States [751]. It is not patented in any other country. PKP licenses the patent, along with other public-key cryptography patents. Anyone interested in obtaining a license should contact:

Robert B. Fougner
Director of Licensing
Public Key Partners
130 B Kifer Court
Sunnyvale, CA 94086
Tel: (408) 735-6779

The U.S. patent will expire on September 20, 2000.

12.5 POHLIG-HELLMAN

The Pohlig-Hellman encryption scheme [703] is similar to RSA. It is not a symmetric algorithm, because different keys are used for encryption and decryption.

It is not a public-key scheme. Both the encryption and decryption keys must be kept secret.

Like RSA,

$$C = M^e \bmod n$$
$$M = C^d \bmod n$$

in which

$$ed = 1 \bmod \text{a complicated number}$$

Unlike RSA, n is not defined in terms of two large primes; it must remain part of the secret key. If someone had e and n, they could calculate d. Without knowledge of e or d, an adversary would be forced to calculate

$$e = \log_M C \bmod n$$

We have already seen that this is a hard problem.

Patents

The Pohlig-Hellman algorithm is patented in the United States [430] and also in Canada. PKP licenses the patent, along with other public-key cryptography patents. Anyone interested in obtaining a license should contact:

Robert B. Fougner
Director of Licensing
Public Key Partners
130 B Kifer Court
Sunnyvale, CA 94086
Tel: (408) 735-6779

12.6 RABIN

Rabin's scheme [724,880] gets its security from the difficulty of finding square roots modulo a composite. This problem is equivalent to factoring.

First choose two primes, p and q, both congruent to 3 mod 4. These primes are the private key; $n = pq$ is the public key.

To encrypt a message, M (M must be less than n), simply compute

$$C = M^2 \bmod n$$

Decrypting the message is just as easy, but slightly more annoying. Since the receivers know p and q, they can solve the two congruences using the Chinese remainder theorem. Compute:

$$M_1 = C^{(p+1)/4} \bmod n$$
$$M_2 = p - C^{(p+1)/4} \bmod n$$
$$M_3 = C^{(q+1)/4} \bmod n$$
$$M_4 = q - C^{(q+1)/4} \bmod n$$

One of those four results, M_1, M_2, M_3, or M_4, equals M. If the message is English text, it should be easy to choose the correct M. On the other hand, if the message is a random bit stream (say, for key generation) or a digital signature, there is no way to determine which M_i is correct.

Consequently, Rabin did not propose this as a public-key cryptography algorithm. However, ElGamal later adapted this idea for his public-key algorithm (see Section 13.3).

Williams

Williams redefined Rabin's schemes to eliminate these shortcomings [880]. In his scheme, p and q are selected such that:

$$p = -1 \pmod 4$$
$$q = -1 \pmod 4$$

and

$$N = p \times q$$

Also, there is a small integer, S, such that $J(S,N) = -1$. ("J" is the Jacobi symbol.) N and S are public. The secret key is k, such that:

$$k = 1/2 \times (1/4 \times (p-1) \times (q-1) + 1)$$

To encrypt a message, M, compute c_1 such that $J(M,N) = (-1)^{a_1}$. Then, compute $M' = S^{c_1} \times M \pmod N$. Like Rabin's scheme, $C = M'^2 \pmod N$. And $c_2 = M' \pmod 2$. The final ciphertext message is the triple:

$$(C,c_1,c_2)$$

To decrypt C, the receiver computes M'' using:

$$C^k = \pm M'' \pmod N$$

The proper sign of M'' is given by c_2. Finally:

$$M = S^{-c_1} \times (-1)^{c_1} \times M'' \pmod N$$

Williams refined this scheme further in [882,883]. Instead of squaring the plaintext message, cube it. The large primes must be congruent to 1 mod 3;

otherwise the public and private keys are the same. Even better, there is only one unique decryption for each encryption.

This algorithm has an advantage over RSA in that it is provably as secure as factoring, and it is completely insecure against a chosen-ciphertext attack. I recommend using RSA.

12.7 FEIGE-FIAT-SHAMIR

Amos Fiat and Adi Shamir's authentication and digital signature scheme is discussed in [348,349]. Uriel Feige, Fiat, and Shamir modified the algorithm to a zero-knowledge proof of identity [332,333]. This is the best-known zero-knowledge proof of identity.

On July 9, 1986, the three authors submitted a U.S. patent application [799]. Because of potential military applications, the application was reviewed by the military. The patent office had six months to respond with a secrecy order. On January 6, 1987, three days before the end of the six-month period, the patents and trademarks office imposed the order at the request of the army. They stated that ". . . the disclosure or publication of the subject matter . . . would be detrimental to the national security. . . ." The authors were ordered to notify all Americans to whom the research had been disclosed that unauthorized disclosure could lead to two years imprisonment, a ten-thousand-dollar fine, or both. Furthermore, the authors had to inform the commissioner of patents and trademarks of all foreign nationals to whom the information had been disclosed.

This was ludicrous. All through the second half of 1986, the authors had presented the work at conferences throughout Israel, Europe, and the United States. The authors weren't even American citizens; all the work had been done at the Weizmann Institute in Israel.

Word of this spread through the academic community and the press. Within two days the secrecy order was rescinded; Shamir and others believe that the NSA pulled strings to rescind the order, although they officially had no comment. Further details of this story are in [528].

Simplified Feige-Fiat-Shamir Identification Scheme

Before issuing any private keys, the arbitrator chooses a random modulus, n, which is the product of two large primes. In real life, n should be at least 512 bits long and probably closer to 1024 bits. This n can be shared among a group of provers.

To generate Peggy's public and private keys, a trusted arbitrator chooses a number, v, when v is a quadratic residue mod n. In other words, choose v such that $x^2 = v \bmod n$ has a solution and $v^{-1} \bmod n$ exists. This is the public key. Then calculate the smallest s for which $s = \text{sqrt}(1/v) \bmod n$. This is the private key.

The identification protocol can now proceed:

(1) Peggy picks a random r, where r is less than n. She then computes $x = r^2 \bmod n$, and sends x to Victor.

(2) Victor sends Peggy a random bit, b.

(3) If $b = 0$, then Peggy sends Victor r. If $b = 1$, then Peggy sends Victor $y = r \times s \mod n$.

(4) If $b = 0$, Victor verifies that $x = r^2 \mod n$, proving that Peggy knows sqrt(x). If $b = 1$, Victor verifies that $x = y^2 \times v \mod n$, proving that Peggy knows sqrt(x/v).

This protocol is a single **accreditation.** Peggy and Victor repeat this protocol t times, until Victor is convinced that Peggy knows s. This is a cut-and-choose protocol. If Peggy doesn't know s, she can pick r such that she can fool Victor if he sends her a 0, or she can pick r such that she can fool Victor if he sends her a 1. She can't do both. The odds of her fooling Victor once is 50%. The odds of her fooling him t times are 1 in 2^t.

Another way of attacking the protocol would be for Victor to try to impersonate Peggy. He could initiate the protocol with another verifier, Valerie. In step (1), instead of choosing a random r, he would just reuse an old r that he saw Peggy use. However, the odds of Valerie's choosing the same value for b in step (2) that Victor did in the protocol with Peggy are 1 in 2. So, the odds of his fooling Valerie are 50%. The odds of his fooling her t times are 1 in 2^t.

For this to work it is imperative that Peggy not reuse an r. If she did, and Victor sent Peggy the other random bit in step (2), then he would have both of Peggy's responses. All he has to do is to collect t of these, and then he can impersonate her.

Feige-Fiat-Shamir Identification Scheme

In their papers [332,333], Feige, Fiat, and Shamir show how parallel construction can increase the number of accreditations per round and reduce Peggy and Victor's interactions.

First generate n as above, as the product of two large primes. To generate Peggy's public and private keys, first choose k different numbers: $v_1, v_2 \ldots v_k$, where v_i is a quadratic residue mod n. In other words, choose v_i such that $x^2 = v_i \mod n$ has a solution. This string, $v_1, v_2 \ldots v_k$, is the public key. Then calculate the smallest s_i such that $s_i = \text{sqrt}(1/v_i) \mod n$. This string, $s_1, s_2, \ldots s_k$, is the private key.

And the protocol is:

(1) Peggy picks a random r, where r is less than n. She then computes $x = r^2 \mod n$, and sends x to Victor.

(2) Victor sends Peggy a random binary string k-bits long: $b_1, b_2, \ldots b_k$.

(3) Peggy computes $y = r \times (s_1 b_1 \times s_2 b_2 \times \ldots \times s_k b_k) \mod n$. (She multiplies the values of s together based on the random binary string Victor sent her. If Victor's first bit is a 1, then s_1 is multiplied; if Victor's first bit is a 0, then s_1 is not multiplied.) She sends y to Victor.

(4) Victor verifies that $x = y^2 \times (v_1 b_1 \times v_2 b_2 \times \ldots \times v_k b_k) \mod n$. (He multiplies the values of s together based on the random binary string. If his first bit is a 1, then s_1 is multiplied; if his first bit is a 0, then s_1 is not multiplied.)

Peggy and Victor repeat this protocol t times, until Victor is convinced that Peggy knows $s_1, s_2, \ldots s_k$.

The chance that Peggy can fool Victor is 1 in 2^{kt}, where t is the number of times they repeat the protocol. They recommend a 1 in 2^{20} chance of a cheater's fooling Victor and suggest that $k = 5$ and $t = 4$. If you are more paranoid, increase these numbers.

An Example

Let's look at this protocol in action with small numbers.

If $n = 35$ (the two primes are 5 and 7), then the possible quadratic residues are:

1: $x^2 = 1 \bmod 35$ has a solution: $x = 1, 6, 29$, or 34.
4: $x^2 = 4 \bmod 35$ has a solution: $x = 2, 12, 23$, or 33.
9: $x^2 = 9 \bmod 35$ has a solution: $x = 3, 17, 18$, or 32.
11: $x^2 = 11 \bmod 35$ has a solution: $x = 9, 16, 19$, or 26.
14: $x^2 = 14 \bmod 35$ has a solution: $x = 7$ or 28.
15: $x^2 = 15 \bmod 35$ has a solution: $x = 15$ or 20.
16: $x^2 = 16 \bmod 35$ has a solution: $x = 4, 11, 24$, or 31.
21: $x^2 = 21 \bmod 35$ has a solution: $x = 14$ or 21.
25: $x^2 = 25 \bmod 35$ has a solution: $x = 5$ or 30.
29: $x^2 = 29 \bmod 35$ has a solution: $x = 8, 13, 22$, or 27.
30: $x^2 = 30 \bmod 35$ has a solution: $x = 10$ or 25.

The inverses (mod 35) and their square roots are:

v	v^{-1}	$S=\sqrt{v^{-1}}$
1	1	1
4	9	3
9	4	2
11	16	4
16	11	9
29	29	8

Note that 14, 15, 21, 25, and 30 do not have inverses mod 35, because they are not relatively prime to 35. This makes sense, because there should be $(5-1) \times (7-1)/4$ quadratic residues mod 35 (see Section 9.3).

So, Peggy gets the public key consisting of $k = 4$ values: {4,11,16,29}. The corresponding private key is {3,4,9,8}. Here's one round of the protocol:

(1) Peggy chooses a random r=16, computes $16^2 \bmod 35 = 11$, and sends it to Victor.

(2) Victor sends Peggy a random binary string {1,1,0,1}.

(3) Peggy computes $16 \times ((3^1) \times (4^1) \times (9^0) \times (8^1)) \bmod 35 = 31$ and sends it to Bob.

(4) Bob verifies that $31^2 \times ((4^1) \times (11^1) \times (16^0)) \times (29^1) \bmod 35 = 11$.

Peggy and Victor repeat the protocol, each time with a different random r, until Victor is satisfied.

With small values like these, there's no real security. But when n is 512 bits long or more, Victor cannot learn anything about Peggy's secret key except the fact that she knows it.

Enhancements

It is possible to embed identification information into the protocol. Assume that I is a binary string representing Peggy's identification: her name, address, social security number, hat size, preferred brand of soft drink, etc. Use a one-way hash function $H(x)$ to compute $H(I,j)$, where j is a small random number concatenated onto I. Find a set of j's where $H(I,j)$ is a quadratic residue mod n. These j's become $v_1, v_2 \ldots v_k$. Peggy's public key is now I and the list of j's. She sends I and the list of j's to Victor before step (1) of the protocol, and Victor generates $v_1, v_2, \ldots v_k$ from $H(I,j)$.

Now, after Victor has successfully completed the protocol with Peggy, he is assured that someone knowing the factorization of the modulus has certified the association between I and Peggy by giving her the square roots of the v_i derived from I.

The authors Feige-Fiat-Shamir also include the following implementation remarks:

> For nonperfect hash functions, it may be advisable to randomize I by concatenating it with a long random string, R. This string is chosen by the arbitrator and is revealed to Victor along with I.
>
> In typical implementations, k should between 1 and 18. Larger values of k can reduce the time and communication complexity by reducing the number of rounds.
>
> The value n should be at least 512 bits long.
>
> If all users choose their own n and publish it in a public key file, they can dispense with the arbitrator. However, this RSA-like variant makes the scheme considerably less convenient.

Fiat-Shamir Signature Scheme

Turning this identification scheme into a signature scheme is basically a matter of turning Victor into a hash function. The primary benefit of the Fiat-Shamir digital signature scheme over RSA is that the typical signature requires only 1% to 4% of the modular multiplications. Therefore, it is a lot faster. For this protocol, we'll use the names Alice and Bob.

The setup is the same as the identification scheme. Choose n to be the product of two large primes. Generate the public key, $v_1, v_2, \ldots v_k$, and the private key, $s_1, s_2, \ldots s_k$, such that $s_i = \text{sqrt}(1/v_i) \bmod n$.

(1) Alice picks t random integers fewer than n: $r_1, r_2, \ldots, r_t$, and computes $x_1, x_2, \ldots, x_t$ such that $x_i = r_i^2 \bmod n$.

(2) Alice hashes the concatenation of the message and the string of x's to generate a bit stream: $H(m, x_1, x_2, \ldots x_t)$. She uses the first $k \times t$ bits of this string as values of b_{ij}, when i goes from 1 to t, and j goes from 1 to k.

(3) Alice computes $y_1, y_2, \ldots, y_t$, where

$$y_i = r_i \times (s_1 b_{i1} \times s_2 b_{i2} \times \ldots \times s_j b_{ij} \times s_k b_{ik}) \bmod n$$

(She multiplies the values of s together based on the random b_{ij} values. If b_{i1} is a 1, then s_1 is multiplied; if b_{i1} is a 0, then s_1 is not multiplied.)

(4) Alice sends Bob m, all the bit values of b_{ij}, and all the values of y_i. He already has Alice's public key: $v_1, v_2, \ldots, v_k$.

(5) Bob computes $z_1, z_2, \ldots, z_t$, where

$$z_i = y_i^2 \times (v_1 b_{i1} \times v_2 b_{i2} \times \ldots v_j b_{ij} \times v_k b_{ik}) \bmod n$$

(Again, Bob multiplies based on the b_{ij} values. Note that z_i should be equal to x_i.)

(6) Bob verifies that the first $k \times t$ bits of $H(m, z_1, z_2, \ldots z_t)$ are the b_{ij} values that Alice sent him.

As with the identification scheme, the security of this signature scheme is proportional to $1/2^{kt}$. It is also related to the difficulty of factoring n. Fiat and Shamir pointed out that forging a signature is easiest when the complexity of factoring n is considerably higher than 2^{kt}. And, because of birthday-type attacks (see Section 14.1), they recommend that $k \times t$ be increased from 20 to at least 72. They suggest $k = 9$ and $t = 8$.

Improved Fiat-Shamir Signature Scheme

Silvio Micali and Adi Shamir improved the Fiat-Shamir protocol in [601]. They chose $v_1, v_2, \ldots v_k$ to be the first k prime numbers. So

$$v_1 = 2, v_2 = 3, v_3 = 5, \text{ etc.}$$

This is the public key.

The private key, $s_1, s_2, \ldots s_k$ is a random square root, determined by

$$s_i = \sqrt{v_i^{-1}} \bmod n$$

In this version, every person must have a different n. The modification makes it easier to verify signatures. The time required to generate signatures, and the security of those signatures, is unaffected.

Other Enhancements

There is also an N-party identification scheme, based on the Fiat-Shamir algorithm [165]. Two other improvements to the Fiat-Shamir scheme are proposed in [679].

Ohta-Okamoto Identification Scheme

This protocol is a modification of Feige-Fiat-Shamir and gets its security from the difficulty of factoring [664,665]. The same authors also wrote a multisignature scheme, by which a number of different people can sequentially sign a message [666]. This scheme has been proposed for smart-card implementation [484].

Patents

Fiat-Shamir is patented [799]. Anyone interested in licensing the algorithm should contact:

Yeda Research and Development
The Weizmann Institute of Science
Rehovot 76100
Israel

More Public-Key Algorithms

13.1 GUILLOU-QUISQUATER

Fiat-Shamir was the first practical zero-knowledge proof of identity. It minimized the computation required in the exchange by increasing the number of iterations and accreditations per iteration. For some implementations, like smart cards, this is less than ideal. Exchanges with the outside world are time-consuming, and the storage required for each accreditation can strain the limited resources of the card.

Guillou and Quisquater developed a zero-knowledge identification algorithm more suited to applications like these [401,721]. The number of exchanges between Peggy and Victor, and the number of parallel accreditations in each exchange, are both kept to an absolute minimum: there is only one exchange of one accreditation for each proof. The computation required by Guillou-Quisquater is more than by Fiat-Shamir for the same level of security, by a factor of three.

Like Fiat-Shamir, this identification algorithm can be converted to a digital signature algorithm. For the same reasons as discussed earlier, it is also suited for smart-card applications.

Guillou-Quisquater Identification Scheme

Peggy is a smart card who wants to prove her identity to Victor. Peggy's identity consists of a set of credentials: a data string consisting of the card's name, validity period, a bank account number, and whatever else the application warrants. This bit string is called J. (Actually, the credentials can be a longer string, and hashed to a J value. This complexity does not modify the protocol in any way, however.) This is analogous to the public key. Other public information, shared by all "Peggys" who could use this application, is an exponent v and a modulus n, when

n is the product of two secret primes. The private key is B, calculated such that $JB^v \equiv 1 \pmod{n}$.

Peggy sends Victor her credentials, J. Now, she wants to prove to Victor that those credentials are hers. To do this, she has to convince Victor that she knows B. Here's the protocol:

(1) Peggy picks a random integer, r, such that r is between 1 and $n-1$. She computes $T \equiv r^v \pmod{n}$ and sends it to Victor.

(2) Victor picks a random integer, d, such that d is between zero and $v-1$. He sends d to Peggy.

(3) Peggy computes $D \equiv rB^d \pmod{n}$ and sends it to Victor.

(4) Victor computes $T' = D^v J^d \pmod{n}$. If $T \equiv T' \pmod{n}$, then the authentication succeeds.

The math isn't that complex:

$$T' = D^v J^d = (rB^d)^v J^d = r^v B^{dv} J^d = r^v (JB^v)^d = r^v \equiv T \pmod{n}$$

since B was constructed to satisfy:

$$JB^v \equiv 1 \pmod{n}$$

Guillou-Quisquater Signature Scheme

This identification can be converted to a signature scheme, also suited for smart-card implementation [402,403].

The public and private key setup is the same as before. Here's the protocol:

(1) Alice picks a random integer, r, such that r is between 1 and $n-1$. She computes $T \equiv r^v \pmod{n}$.

(2) Alice computes $d = \mathrm{H}(M,T)$, where M is the message being signed and $\mathrm{H}(x)$ is a one-way hash function.

(3) Alice computes $D \equiv rB^d \pmod{n}$. The signature consists of the message, M, the two calculated values, d and D, and her credentials, J.

(4) Bob computes $T' = D^v J^d \pmod{n}$. He then computes $d' = \mathrm{H}(M,T')$. If $d \equiv d'$, then Alice must know B, and the signature is valid.

Multiple Signatures

What if many people wanted to sign the same document? There is an easy solution—each of them signs separately—but this signature scheme can do better than that. This example has Alice and Bob signing the same document and Carol verifying the signatures, but any number of people can be involved in the signature process. As before, Alice and Bob have their own unique J and B values: J_A and J_B, and B_A and B_B. The values n and v are common to the system.

(1) Alice picks a random integer, r_A, such that r_A is between 1 and $n-1$. She computes $T_A \equiv r_A^{\,v} \pmod{n}$.

(2) Bob picks a random integer, r_B, such that r_B is between 1 and $n-1$. He computes $T_B \equiv r_B^{\,v} \pmod{n}$.

(3) Alice and Bob compute $T \equiv T_A T_B \pmod{n}$.

(4) Alice and Bob compute $d = \mathrm{H}(M,T)$, when M is the message being signed and $\mathrm{H}(x)$ is a one-way hash function.

(5) Alice computes $D_A \equiv r_A B_A^{\,d} \pmod{n}$.

(6) Bob computes $D_B \equiv r_B B_B^{\,d} \pmod{n}$.

(7) Alice and Bob compute $D \equiv D_A D_B \pmod{n}$. The signature consists of the message, M, the two calculated values, d and D, and both of their credentials: J_A and J_B.

(8) Carol computes $J \equiv J_A J_B$.

(9) Carol computes $T' = D^v J^d \pmod{n}$. She then computes $d' = \mathrm{H}(M,T')$. If $d \equiv d'$, then the multiple signature is valid.

This protocol can be extended to any number of people. For multiple people to sign, they all multiply their individual T_i values together in step (3) and their individual D_i values together in step (7). To verify a multiple signature, multiply all the signers' J_i values together in step (8). In banking, this ability for multiple people to cosign a document could be useful.

13.2 ONG-SCHNORR-SHAMIR

Ong, Schnorr, and Shamir developed a class of signature schemes using polynomials modulo n [680,681].

Choose a large integer n (it is not necessary to know the factorization of n). Then choose a random integer, k, such that k and n are relatively prime. Calculate h such that:

$$h = -k^{-2} \pmod{n} = -(k^{-1})^2 \pmod{n}$$

The public key is h and n; k is the private key.

To sign a message, M, first generate a random number, r, such that r and n are relatively prime. Then calculate:

$$S_1 = 1/2 \times (M/r + r) \pmod{n}$$
$$S_2 = k/2 \times (M/r - r) \pmod{n}$$

The pair, S_1 and S_2, is the signature.

To verify a signature, confirm that:

$$M \equiv S_1^{\,2} + h \times S_2^{\,2} \pmod{n}$$

The version of the scheme described here is based on quadratic polynomials. When it was first proposed in [678], a $100 reward was offered for successful cryptanalysis. Pollard proved it insecure [705], but its authors were not deterred. They proposed a modification of the algorithm based on cubic polynomials, which Pollard also broke [705]. The authors then proposed a quartic version [705], which was also broken [319,705]. There have been no more schemes of this type proposed since then.

13.3 ELGAMAL

The ElGamal scheme [316], first presented in [315], is a variant of Rabin (see Section 12.6). It can be used for both digital signatures and encryption and gets its security from the difficulty of calculating discrete logarithms.

To generate a key pair, first choose a prime, p, and two random numbers, g and x, such that both g and x are less than p. Then calculate

$$y = g^x \pmod{p}$$

The public key is y, g, and p. Both g and p can be shared among a group of users. The private key is x.

ElGamal Signatures

To sign a message, M, first choose a random number, k, such that k is relatively prime to $p-1$. Then compute

$$a = g^k \pmod{p}$$

and use the extended Euclidean algorithm to solve for b in the following equation:

$$M = xa + kb \pmod{p-1}$$

The signature is the pair: a and b. The random variable, k, must be kept secret.

To verify a signature, confirm that

$$y^a a^b \pmod{p} = g^M \pmod{p}$$

This is summarized in Table 13.1.

For example, choose $p = 11$ and $g = 2$. Choose private key $x = 8$. Calculate:

$$y = g^x \pmod{p} = 2^8 \pmod{11} = 3$$

The public key is $y = 3$, $g = 2$, and $p = 11$.

TABLE 13.1
ElGamal Signatures

PUBLIC KEY:

p prime (can be shared among a group of users)

g $<p$ (can be shared among a group of users)

y $= g^x \pmod p$

PRIVATE KEY:

x $<p$

SIGNING:

k choose at random, relatively prime to $p-1$.

a (signature) $= g^k \pmod p$

b (signature) such that $M = xa + kb \pmod{p-1}$

VERIFYING:

Accept as valid if $y^a a^b \pmod p = g^M \pmod p$

To authenticate $M = 5$, first choose a random number, $k = 9$. Confirm that gcd(9,10) = 1. Compute:

$$a = g^k \pmod p = 2^9 \pmod{11} = 6$$

and use the Euclidean algorithm to solve for b:

$$M = ax + kb \pmod{p-1}$$
$$5 = 8 \times 6 + 9 \times b \pmod{10}$$

The solution is $b = 3$ and the signature is the pair: $a = 6$ and $b = 3$.

To verify a signature, confirm that

$$y^a a^b \pmod p = g^M \pmod p$$
$$3^6 6^3 \pmod{11} = 2^5 \pmod{11}$$

Since the math is all correct, they do this step.

Thomas Beth invented a variant of the ElGamal scheme suitable for proofs of identity [90].

ElGamal Encryption

A modification of ElGamal can encrypt messages. To encrypt message M, first choose a random k, such that k is relatively prime to $p-1$. Then compute:

$$a = g^k \bmod p$$
$$b = y^k M \bmod p$$

The pair, a and b, is the ciphertext. Note that the ciphertext is twice the size of the plaintext.

To decrypt a and b, compute:

$$b/a^x \pmod{p}$$

Since $a^x \equiv g^{kx} \bmod p$, and $b/a^x \equiv y^k M/a^x \equiv g^{xk} M/g^{xk} \equiv M \bmod p$, this all works (see Table 13.2).

TABLE 13.2
ElGamal Encryption

PUBLIC KEY:	
p	prime (can be shared among a group of users)
g	$< p$ (can be shared among a group of users)
y	$= g^x \pmod{p}$
PRIVATE KEY:	
x	$< p$
ENCRYPTING:	
k	choose at random, relatively prime to $p-1$
a (ciphertext)	$= g^k \pmod{p}$
b (ciphertext)	$= y^K M \pmod{p}$
DECRYPTING:	
M (plaintext)	$= b/a^x \pmod{p}$

Patents

ElGamal is unpatented. But, before you go ahead and implement the algorithm, realize that PKP feels that this algorithm is covered under the Diffie-Hellman patent [426].

13.4 SCHNORR

Schnorr's authentication and signature scheme [776,777] combines ideas from ElGamal [316], Fiat-Shamir [348], and David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graaf's interactive protocol [205,204]. It gets its security from the difficulty of calculating discrete logarithms.

To generate a key pair, first choose two primes, p and q, such that q is a prime factor of $p-1$. Then, choose an a not equal to 1, such that $a^q = 1 \bmod p$. All these numbers can be common to a group of users and can be freely published.

To generate a particular public/private key pair, choose a random number less than q. This is the private key, s. Then calculate $v = a^{-s} \bmod p$. This is the public key.

Authentication Protocol

(1) Peggy picks a random number, r, less than q, and computes $x = a^r \bmod p$. This is the preprocessing stage and can be done long before Victor is present.

(2) Victor sends Peggy a random number, e, between 0 and 2^t-1. (I'll discuss t in a moment.)

(3) Peggy computes $y = r + s \times e \bmod q$ and sends y to Victor.

(4) Victor verifies that $x = a^y v^e \bmod p$.

The security is based on the parameter t. The difficulty of breaking the algorithm is about 2^t. Schnorr recommends that p be about 512 bits and q be about 140 bits. The value of t that is commensurate with this kind of security is $t = 72$, which Schnorr feels is secure enough.

Digital Signature Protocol

The private/public key pair is the same, but we're now adding a one-way hash function, H(M).

(1) Alice picks a random number, r, less than q, and computes $x = a^r \bmod p$. This computation is the preprocessing stage.

(2) Alice concatenates the message and x, and hashes the result:

$$e = \mathrm{H}(m,x)$$

Alice computes $y = r + s \times e \bmod q$. The signature is e and y; she sends these to Bob.

(3) Bob computes $z = a^y v^e \bmod p$. He then confirms that the concatenation of the message and z hashes to e:

$$e = \mathrm{H}(m,z)$$

If they do, he accepts the signature as valid.

In his paper, Schnorr cites these novel features of his algorithm:

> Most of the computation for signature generation can be completed in a preprocessing stage, independent of the message being signed. Hence, it can be done during idle time and not affect the signature speed. An attack against this preprocessing stage is discussed in [279], but I don't think it's practical.
>
> For the same level of security, the length of signatures is less for Schnorr than for RSA. For example, with a 140-bit q, signatures are only 212-bits

long, less than half the length of RSA signatures. Schnorr's signatures are also much shorter than ElGamal's.

A modification of this algorithm, by Ernie Brickell and Kevin McCurley, enhances its security. It is in [166].

Patents

Schnorr is patented in the United States [778] and in many other countries. In 1993, PKP acquired the worldwide rights to the patent. The U.S. patent expires on February 19, 2008.

13.5 DIGITAL SIGNATURE ALGORITHM (DSA)

In August 1991, The National Institute of Standards and Technology (NIST) proposed the Digital Signature Algorithm (DSA) for use in their new Digital Signature Standard (DSS). According to the *Federal Register* [329]:

> A Federal Information Processing Standard (FIPS) for Digital Signature Standard (DSS) is being proposed. This proposed standard specifies a public-key digital signature algorithm (DSA) appropriate for Federal digital signature applications. The proposed DSS uses a public key to verify to a recipient the integrity of data and identity of the sender of the data. The DSS can also be used by a third party to ascertain the authenticity of a signature and the data associated with it.
>
> This proposed standard adopts a public-key signature scheme that uses a pair of transformations to generate and verify a digital value called a signature.

And:

> This proposed FIPS is the result of evaluating a number of alternative digital signature techniques. In making the selection NIST has followed the mandate contained in section 2 of the Computer Security Act of 1987 that NIST develop standards to ". . . assure the cost-effective security and privacy of Federal information and, among technologies offering comparable protection, on selecting the option with the most desirable operating and use characteristics.
>
> Among the factors that were considered during this process were the level of security provided, the ease of implementation in both hardware and software, the ease of export from the U.S., the applicability of patents, impact on national security and law enforcement and the level of efficiency in both the signing and verification functions.
>
> A number of techniques were deemed to provide appropriate protection for Federal systems. The technique selected has the following desirable characteristics:

—NIST expects it to be available on a royalty-free basis. Broader use of this technique resulting from public availability should be an economic benefit to the government and the public.

—The technique selected provides for efficient implementation of the signature operations in smart card applications. In these applications the signing operations are performed in the computationally modest environment of the smart card while the verification process is implemented in a more computationally rich environment such as a personal computer, a hardware cryptographic module, or a mainframe computer.

Before it gets too confusing, let me review the nomenclature: DSA is the algorithm; the DSS is the standard. The standard employs the algorithm. The algorithm is part of the standard. (NIST claimed that having DES meant that both the algorithm and the standard were too confusing—I'm not sure this is much better.)

Reaction to the Announcement

NIST's announcement created a maelstrom of criticisms and accusations. Unfortunately, it was more political than academic. RSA Data Security, Inc., purveyors of the RSA algorithm, led the criticism against DSS. They wanted RSA, not another algorithm, used as the standard. RSADSI makes a lot of money licensing the RSA algorithm, and a royalty-free digital signature standard would directly affect their bottom line. (Note: DSA is not necessarily free of any patent infringements; I'll discuss that later.)

Before the algorithm was announced, RSADSI campaigned against a "common modulus," which might have given the government the ability to forge signatures. When the algorithm was announced without this common modulus, they attacked it on other grounds [97], both in letters to NIST and statements to the press. (Four letters to NIST appeared in [747]. When reading them, keep in mind that at least two of the authors, R. Rivest and M. Hellman, have a financial interest in DSS's not being approved.)

Many large software companies that already licensed the RSA algorithm came out against the DSS. In 1982, the government solicited public-key algorithms for a standard [328]. After that, there wasn't a peep out of NIST for nine years. Companies such as IBM, Apple, Novell, Lotus, Northern Telecom, Microsoft, DEC, and Sun had already spent large amounts of money implementing the RSA algorithm. They were not interested in losing their investment.

In all, NIST received 109 comments by the end of the first comment period on February 28, 1992.

Let's look at the criticisms against DSA, one by one:

1. DSA cannot be used for encryption or key distribution.

   This is true, but that is not the point of the standard. It is a signature standard. There should be a NIST standard for public-key encryption. NIST is committing

a grave injustice to the American people by not implementing a public-key encryption standard. It is suspicious that NIST proposed a digital signature standard that cannot be used for encryption. (As it turns out, though, it can—see Section 16.6.) That does not mean that a signature standard is useless.

2. DSA was developed by the NSA, and there may be a trap door in the algorithm.

   Much of the initial comments stemmed from cryptographers' paranoia: "NIST's denial of information with no apparent justification does not inspire confidence in DSS, but intensifies concern that there is a hidden agenda, such as laying the groundwork for a national public-key cryptosystem that is in fact vulnerable to being broken by NIST and/or NSA" [97]. There was one serious question about the security of DSS by Arjen Lenstra and Stuart Haber at Bellcore. This will be discussed later.

3. DSA is slower than RSA [466].

   This is true, more or less. Signature generation speeds are the same, but signature verification can be 10 to 40 times slower with DSA. Key generation is faster than RSA. Key generation is irrelevant; a user rarely does that. On the other hand, signature verification is the most common operation.

   The problem with this criticism is that there are many ways to play with the test parameters, depending on the results you want. Precomputations can speed up DSA signature generation, but they are not always applicable. Proponents of RSA use numbers optimized to make their calculations easier; proponents of DSA use their own optimizations. In any case, computers are getting faster all the time. While there is a difference, in most applications it will not be noticeable.

4. RSA is a de facto standard.

   These are two examples of the complaint. From Robert Follett, the program director of standards at IBM [351]:

   > IBM is concerned that NIST has proposed a standard with a different digital signature scheme rather than adopting the international standard. We have been convinced by users and user organizations that the international standards using RSA will be a prerequisite to the sales of security products in the very near future.

   From Les Shroyer, vice president and director, corporate MIS and telecommunications, at Motorola [813]:

   > We must have a single, robust, politically-accepted digital signature standard that is usable throughout the world, between both US and non-US, and Motorola and non-Motorola entities. The lack of other viable digital signature technology for the last eight years has made RSA a de facto standard. . . . Motorola and many other companies . . .

have committed millions of dollars to RSA. We have concern over the interoperability and support of two different standards, as that situation will lead to added costs, delays in deployment, and complication. . . .

Many companies wanted NIST to adopt the ISO/IEC 9796, the international digital signature standard that uses RSA [446]. While this is a valid complaint, it is not a sufficient justification to make it a standard. A royalty-free standard would serve the U.S. public interest to a much greater degree.

5. The DSA selection process was not public; sufficient time for analysis has not been provided.

   First NIST claimed that they designed the DSA; then they admitted that NSA helped them. Finally, they confirmed that NSA designed the algorithm. This worries many people; the NSA doesn't inspire trust. Even so, the algorithm is public and available for analysis; and NIST extended the time for analysis and comment.

6. DSA may infringe on other patents.

   It may. This will be discussed in the section on patents.

7. The key size is too small.

   This was the only valid criticism of DSS. The original implementation set the modulus at 512 bits [650]. Since the algorithm gets its security from the difficulty of computing discrete logs in that modulus, this was worrisome to most cryptographers. There have been advances in the problem of calculating discrete logarithms in a finite field, and 512 bits is too short for long-term security. According to Brian LaMacchia and Andrew Odlyzko, ". . . even 512-bit primes appear to offer only marginal security. . . ." [526]. This isn't truly fair—512 bits provide sufficient security today. But if your security needs stretch into the far future, it's not secure. In response to this criticism, NIST made the key size variable, from 512 bits to 1024 bits. This will be discussed in more detail later.

Description of the Algorithm

DSA is a variant of the Schnorr and ElGamal signature algorithms, and is fully described in [651]. The algorithm uses the following parameters:

p = a prime number 2^L bits long, where L ranges from 512 to 1024 and is a multiple of 64. (In the original standard, the size of p was fixed at 512 bits [650]. This was the source of much criticism and was changed by NIST [651].)

q = a 160-bit prime factor of $p-1$.

$g = h^{(p-1)/q}$, when h is any number less than $p-1$ such that $h^{(p-1)/q} \bmod p$ is greater than 1.

x = a number less than q.

$y = g^x \bmod p$.

Additionally, the algorithm makes use of a one-way hash function: $H(x)$. For the DSS, this is the Secure Hash Algorithm, discussed in Section 14.7.

The first three parameters, p, q, and g, are public and can be common across a network of users. The private key is x; the public key is y.

To sign a message, m:

(1) Alice generates a random number, k, less than q.

(2) Alice generates

$$r = (g^k \bmod p) \bmod q$$
$$s = (k^{-1}(H(m) + xr)) \bmod q$$

The parameters r and s are her signature; she sends these to Bob.

(3) Bob verifies the signature by computing:

$$w = s^{-1} \bmod q$$
$$u1 = (H(m) \times w) \bmod q$$
$$u2 = (r \times w) \bmod q$$
$$v = ((g^{u1} \times y^{u2}) \bmod p) \bmod q$$

If $v = r$, then the signature is verified.

Proofs for the mathematical relationships are found in the documentation. See Table 13.3 for a summary.

TABLE 13.3
DSA Signatures

PUBLIC KEY:	
p	512-bit to 1024-bit prime (can be shared among a group of users)
q	160-bit prime factor of $p-1$ (can be shared among a group of users)
g	$h^{(p-1)/q}$, where h is less than $p-1$ and $h^{(p-1)/q} \bmod q > 1$ (can be shared among a group of users)
y	$= g^x \pmod p$ (a 160-bit number)
PRIVATE KEY:	
x	$< q$ (a 160-bit number)
SIGNING:	
k	choose at random, less than q
r (signature)	$= (g^k \bmod p) \bmod q$
s (signature)	$= (k^{-1}(H(m) + xr)) \bmod q$
VERIFYING:	
w	$= s^{-1} \bmod q$
$u1$	$= (H(m) \times w) \bmod q$
$u2$	$= (r \times w) \bmod q$
v	$= ((g^{u1} \times y^{u2}) \bmod p) \bmod q$
If $v = r$, then the signature is verified.	

Precomputations

Real-world implementations of DSA can be speeded up through precomputations. Notice that the value r is not dependent on the message. You can create a string of random k values and then precompute r values for each of them. You can also precompute k^{-1} for each of those k values. Then, when a message comes along, you can compute s for a given r and k^{-1}.

This precomputation speeds up the DSA considerably. Table 13.4 is a comparison of DSA and RSA computation times for a smart-card implementation [834]:

TABLE 13.4
Comparison of RSA and DSA Computation Times

	DSA	RSA	DSA with Common p,q,g
Global Computations	Off Card (P)	N/A	Off Card (P)
Key Generation	14 secs.	Off Card (S)	4 secs.
Precomputation	14 secs.	N/A	4 secs.
Signature	.03 secs.	15 secs.	.03 secs.
Verification	16 secs.	1.5 secs.	10 secs.
	1–5 secs. Off Card (P)		1–3 secs. Off Card (P)

Off-card computations were performed on an 80386, 33MHz, personal computer. (P) indicates public parameters off card, and (S) indicates secret parameters off card. Both algorithms use a 512-bit modulus.

DSA Prime Generation

Lenstra and Haber pointed out that certain moduli are much easier to crack than others [535]. If someone used one of these "cooked" moduli, then their signatures would be easy to forge. This isn't a problem for two reasons: first, the moduli for which this property holds true are easy to detect. Second, these moduli are so rare that the chances of using one when choosing a modulus randomly are negligibly small—smaller, in fact, than the chances of accidentally generating a composite number using a probabilistic prime generation routine.

NIST recommends a method for generating the two primes, p and q, such that q divides $p-1$. The prime p is between 512 and 1024 bits long, in some multiple of 64 bits. The prime q is 160 bits long. Let $L-1 = n \times 160 + b$, where L is the length of p, and n and b are two numbers.

(1) Choose an arbitrary sequence of at least 160 bits and call it S. Let g be the length of S in bits.

(2) Compute $U = \text{SHA}(S)$ XOR $\text{SHA}((S+1) \bmod 2^g)$.

(3) Form q from U by setting the most significant bit and the least significant bit to 1.

(4) Check whether q is prime.

(5) If q is not prime, go back to step (1).

(6) Let $C = 0$ and $N = 2$.

(7) For $k = 0,1, \ldots, n$, let $V_k = \text{SHA}((S + N + k) \bmod 2^g)$.

(8) Let

$$W = V_0 + V_1 \times 2^{160} + \ldots + V_{n-1} \times 2^{(n-1)\times 160} + (V_n \bmod 2^b) \times 2^{n\times 160}$$

and

$X = W + 2^{L-1}$. Note that X is a 2^L-bit number.

(9) Let $p = X - ((X \bmod 2q) - 1)$. Note that p is congruent to 1 mod $2q$.

(10) If $p < 2^{L-1}$, then go to step 13.

(11) Check whether p is prime.

(12) If p is prime, go to step (15).

(13) Let $C = C+1$ and $N = N+n+1$.

(14) If $C = 4096$, then go to step (1). Otherwise, go to step (7).

(15) Save the value of S and the value of C used to generate p and q.

The point of this exercise is that there is a public means of generating p and q. Remember the people worried about "cooked" values of p and q that make breaking the DSA easier? For all practical purposes, this method prevents that. If someone hands you a p and a q, you might wonder where they got it. However, if someone hands you a value for S and C that generated the random p and q, you can go through this routine yourself. The use of SHA, a one-way function, prevents someone from working backwards.

This security is better than what you get with RSA. In RSA, the prime numbers are kept secret. Someone could generate a fake prime or one of a special form that makes factoring easier. Unless you know the private key, you won't know that. Here, even if you don't know the private key, you can confirm that p and q have been generated randomly.

ElGamal Encryption with DSA

There have been allegations that the government likes the DSA because it is only a digital signature algorithm and can't be used for encryption. While this is true, it is possible to use a DSA-function call to do ElGamal encryption.

Assume that the DSA algorithm is implemented with a single-function call:

```
DSAsign (p,q,g,k,x,h,r,s)
```

You supply the numbers p, q, g, k, x, and h, and the function returns the signature parameters: r and s.

To do ElGamal encryption of message m with public key y, choose a random number, k, and call:

```
DSAsign (p,p,g,k,0,0,r,s)
```

The value of r returned is a in the ElGamal scheme. Throw s away. Then, call:

```
DSAsign (p,p,y,k,0,0,r,s)
```

Rename the value of r to be u; throw s away. Call:

```
DSAsign (p,p,m,1,u,0,r,s)
```

Throw r away. The value of s returned is b in the ElGamal scheme. You now have the ciphertext, a and b.

Decryption is just as easy. Using secret key x, and ciphertext messages a and b, call:

```
DSAsign (p,p,a,x,0,0,r,s)
```

The value r is $a^x \bmod p$. Call that e. Then call:

```
DSAsign (p,p,1,e,b,0,r,s)
```

The value s is the plaintext message, m.

This method will not work with all implementations of DSA. Some may fix the values or p and q, or the lengths of some of the other parameters. Still, if the implementation is general enough, this is a way to encrypt with a digital signature library.

RSA Encryption with DSA

RSA encryption is even easier. With a modulus n, message m, and a public key e, call:

```
DSAsign (n,n,m,e,0,0,r,s)
```

The value of r returned is the ciphertext.

RSA decryption is the same thing:

```
DSAsign (n,n,m,d,0,0,r,s)
```

returns the plaintext as the value of r.

Security of DSA

At 512-bits, DSA wasn't strong enough for long-term security. At 1024 bits, it is.

The National Security Agency, in its first public interview on the subject, commented to Joe Abernathy of the *Houston Chronicle* on allegations about a trap door in DSS [218]:

> Regarding the alleged trapdoor in the DSS. We find the term trapdoor somewhat misleading since it implies that the messages sent by the DSS are encrypted and with access via a trapdoor one could somehow decrypt (read) the message without the sender's knowledge.
>
> The DSS does not encrypt any data. The real issue is whether the DSS is susceptible to someone forging a signature and therefore discrediting the entire system. We state categorically that the chances of anyone—including NSA—forging a signature with the DSS when it is properly used and implemented is infinitesimally small.
>
> Furthermore, the alleged trapdoor vulnerability is true for *any* public key-based authentication system, including RSA. To imply somehow that this only affects the DSS (a popular argument in the press) is totally misleading. The issue is one of implementation and how one goes about selecting prime numbers. We call your attention to a recent EUROCRYPT conference which had a panel discussion on the issue of trapdoors in the DSS. Included on the panel was one of the Bellcore researchers who initially raised the trapdoor allegation, and our understanding is that the panel—including the person from Bellcore—concluded that the alleged trapdoor was not an issue for the DSS. Furthermore, the general consensus appeared to be that the trapdoor issue was trivial and had been overblown in the press. However, to try to respond to the trapdoor allegation, at NIST's request, we have designed a prime generation process which will ensure that one can avoid selection of the relatively few weak primes which could lead to weakness in using the DSS. Additionally, NIST intends to allow for larger modulus sizes up to 1024 which effectively negates the need to even use the prime generation process to avoid weak primes. An additional very important point that is often overlooked is that with the DSS the primes are *public* and therefore can be subject to public examination. Not all public key systems provide for this same type of examination.
>
> The integrity of any information security system requires attention to proper implementation. With the myriad of vulnerabilities possible given the differences among users, NSA has traditionally insisted on centralized trusted centers as a way to minimize risk to the system. While we have designed technical modifications to the DSS to meet NIST's requests for a more decentralized approach, we still would emphasize that portion of the Federal Register notice for the DSS which states:
>
> > "While it is the intent of this standard to specify general security requirements for generating digital signatures, conformance to this standard does not assure that a particular implementation is secure. The responsible authority in each agency or department shall assure that an overall implementation provides an acceptable level of security. NIST will be working with government users to ensure appropriate implementations."
>
> Finally, we have read all the arguments purporting insecurities with the DSS, and we remain unconvinced of their validity. The DSS has been

> subjected to intense evaluation within NSA which led to its being endorsed by our Director of Information Systems Security for use in signing unclassified data processed in certain intelligence systems and even for signing classified data in selected systems. We believe that this approval speaks to the lack of any credible attack on the integrity provided by the DSS given proper use and implementation. Based on the technical and security requirements of the U.S. government for digital signatures, we believe the DSS is the best choice. In fact, the DSS is being used in a pilot project for the Defense Message System to assure the authenticity of electronic messages of vital command and control information. This initial demonstration includes participation from the Joint Chiefs of Staff, the military services, and Defense Agencies and is being done in cooperation with NIST.

I'm not going to comment on the trustworthiness of the NSA. Take their comment for what you think it's worth.

Subliminal Channel in DSS

Gus Simmons discovered a subliminal channel in DSS [829,830] (see Section 16.7). This subliminal channel allows people to embed a secret message in their signature that can only be read by another person who knows the key. According to Simmons, it is a "remarkable coincidence" that the "apparently inherent shortcomings of subliminal channels using the ElGamal scheme can all be overcome" in the DSS, and that the DSS "provides the most hospitable setting for subliminal communications discovered to date." NIST and NSA have not commented on this subliminal channel; no one knows if they even knew about it. Since this subliminal channel allows an unscrupulous implementer of DSS to leak a piece of the private key with each signature, it is important to never use an implementation of DSS if you don't trust the implementer.

Dangers of a Common Modulus

Even though the DSS does not specify a common modulus to be shared by everyone, different implementations may. For example, the Internal Revenue Service (IRS) is considering using the DSS for the electronic submission of tax returns. What if they require every taxpayer in the country to use a common p and q? Even though the standard doesn't require a common modulus, this implementation accomplishes the same thing. The danger of a common modulus is that it becomes a tempting target. It is still too early to tell what kind of DSS implementations there will be, but there is some cause for concern.

Patents

According to NIST [329]:

> The government has applied to the U.S. Patent Office for a patent on this technique. NIST intends to make this DSS technique available world-wide on a royalty-free basis to the public interest. We believe this technique is patentable and that no other patents would apply to the DSS, but we cannot give firm assurances to such effect in advance of issuance of the patent.

Since then, the U.S. patent office has issued a patent to David Kravitz of NSA [508a]. Even so, three patentholders claim that the DSA infringes on their patents: Diffie-Hellman [426], Merkle-Hellman [428], and Schnorr [778].

The Schnorr patent is the most troublesome. The other two patents expire in 1997; the Schnorr patent is valid until 2008. The Schnorr algorithm was not developed with government money; unlike the PKP patents, the U.S. government has no rights to the Schnorr patent; and Schnorr patented his algorithm worldwide. Even if the U.S. courts rule in favor of DSA, it is unclear what other courts around the world will do. Is an international company going to adopt a standard that may be legal in some countries but infringe on a patent in others? This issue will take time to resolve; at the time of this writing it isn't even resolved in the United States.

In June 1993, NIST proposed to give Public Key Partners an exclusive patent license to DSA [331a]. All other users of DSA would have to pay royalties to PKP. This solution is the worst of both worlds: users won't conform to international signature standards, and users won't be able to implement signatures royalty-free. At this writing, nothing is final. There is at least one lawsuit pending which charges NIST of improper patent licensing [772a].

13.6 ESIGN

ESIGN is a digital signature scheme by T. Okamoto and others at NTT Japan [669,357]. According to its authors, it is at least as secure and considerably faster than either RSA or DSA, with similar key and signature lengths.

The secret key is a pair of large prime numbers, p and q. The public key is n, where

$$n = p^2 q$$

H(x) is a hash function that operates on a message, m. There is also a security parameter, k, which will be discussed shortly.

(1) Alice picks a random number, x, when x is less than $p \times q$

(2) Alice computes:

w, the least integer that is larger than $(\mathrm{H}(m) - x^k \bmod n)/pq$

$s = x + ((w/kx^{k-1}) \bmod p)pq$

Alice sends s to Bob.

(3) To verify the signature, Bob computes $s^k \bmod n$. He also computes a, which is the least integer larger than the number of bits of n divided by 3. If H(m) is less than or equal to $s^k \bmod n$, and if $s^k \bmod n$ is less than H(m) + 2^a, then the signature is considered valid.

This algorithm can be speeded up with precomputation. This precomputation can be done at any time and has nothing to do with the message being signed. After picking x, Alice could break step (2) into two partial steps. The first can be precomputed.

(2a) Alice computes:

$$u = x^k \bmod n$$
$$v = 1/(kx^{k-1}) \bmod p$$

(2b) Alice computes:

$$w = \text{the least integer that is larger than } (H(m) - u)/pq)$$
$$s = x + (w \times v \bmod p) \times p \times q$$

For the size numbers generally used, this precomputation speeds up the signature process by a factor of ten. Almost all the hard work is done in the precomputation stage. A discussion of modular arithmetic operations to speed the ESIGN algorithm can be found in [894,893]. This algorithm can also be extended to work with elliptic curves [670].

Security of ESIGN

When this algorithm was originally proposed, k was set to 2 [677]. This was quickly broken by Ernie Brickell and John DeLaurentis [163], who then extended their attack to $k = 3$. A modified version of this algorithm [667] was broken by Shamir [668]. The variant proposed in [668] was broken in [858]. ESIGN is the current incarnation of this family of algorithms. Another new attack [546] does not work against the ESIGN variant.

The authors currently recommend these values for k: 8, 16, 32, 64, 128, 256, 512, and 1024. They also recommend p and q to be of at least 192 bits each, making n at least 576 bits long. With these parameters, the authors conjecture that ESIGN is as secure as RSA or Rabin.

Patents

ESIGN is patented in the United States [672], Canada (patent number 255,784), England, France, Germany, and Italy. Anyone wishing to license the algorithm should contact:

Intellectual Property Department
NTT
1-6 Uchisaiwai-cho, 1-chome, Chiyada-ku
100 Japan

An ESIGN program for Maxell IC cards, NEC personal computers, and SUN workstations is available through:

Information Systems Technology Department
NTT Advance Technology Corp.
3-2-4 Miharu-cho, Yokasuka-shi
239 Japan
Tel: +81-468-27-0800
Fax: +81-468-23-3558

13.7 McELIECE

McEliece developed a public-key cryptosystem based on algebraic coding theory [580]. The algorithm makes use of the existence of a class of error-correcting codes, known as Goppa codes. His idea was to construct a Goppa code and disguise it as a general linear code. There is a fast algorithm for decoding Goppa codes, but decoding general linear codes is a hard problem. A good description of this algorithm can be found in [687]. The following is a quick summary:

Let $d_H(x)$ denote the Hamming distance. The numbers n, k, and t are system parameters.

The private key has three parts: G', is a $k \times n$ generator matrix for a Goppa code that can correct t errors. P is an $n' \times n$ permutation matrix. S is a $k' \times k$ nonsingular matrix.

The public key is a $k' \times n$ matrix G: $G = SG'P$.

Plaintext messages are strings of k bits, in the form of k-dimensional vectors over GF(2).

To encrypt a message, choose a randomly chosen n-dimensional vector over GF(2), z, with Hamming weight less than or equal to t.

$$c = mG + z$$

To decrypt the ciphertext, first compute $c' = cP^{-1}$. Then, using the decoding algorithm for the Goppa code, find m' such that $d_H(m'G,c')$ is less than or equal to t. Finally, compute $m = m'S^{-1}$.

In his original paper, McEliece suggested that $n = 1024$, $t = 50$, and $k = 524$.

Although the algorithm was one of the first public-key algorithms, and there were no successful cryptanalytic results against the algorithm, it has never gained wide acceptance in the cryptographic community. The scheme is two to three orders of magnitude faster than RSA, but there are several problems. The public key is enormous: 2^{19} bits long. The data expansion is large: the ciphertext is twice as long as the plaintext.

Some attempts at cryptanalysis of this system can be found in [5,532,860]. None of these was successful, although the similarity between the McEliece algorithm and knapsacks worried some.

In 1991, two cryptographers from the former Soviet Union, Korzhik and Turkin [502], broke the McEliece system (one of the benefits of the breakup of the Soviet Union is that ex-Soviet cryptographers are now coming to Western conferences). Their attack requires $20 \times n^3$ operations. It can break the system with the parameters that McEliece originally suggested in about 60 hours.

This scheme is extended in [241]. A variant of this scheme was proposed as a symmetric cryptosystem [730] but was quickly broken [437,848], despite the rebuttal in [729]. At this writing, though, there are still considerable doubts in the cryptologic community about this attack.

13.8 OKAMOTO 92

This signature scheme, first presented by Tatsuaki Okamoto in [670], gets its security from the discrete logarithm problem.

There are two secret primes, p and q, such that q divides $p-1$. The value of q should be at least 140 bits, and the value of p should be at least 512 bits. There are also two random numbers, g_1 and g_2, both the same length as q. The secret key is a pair of random numbers, s_1 and s_2, both less than q. The public key is v, where:

$$v = g_1^{-s_1} g_2^{-s_2} \bmod p$$

To sign a message m, first choose two random numbers, r_1 and r_2, both less than q. Then compute:

$$e = \mathrm{H}(g_1^{r_1} g_2^{r_2} \bmod p, m), \text{ when } \mathrm{H}(x) \text{ is a one-way hash function}$$
$$y_1 = r_1 + es_1 \bmod q$$
$$y_2 = r_2 + es_2 \bmod q$$

The signature is the triple (e, y_1, y_2).

To verify the signature, confirm that e equals:

$$\mathrm{H}(g_1^{y_1} g_2^{y_2} v^e \bmod p, m)$$

If it does, the signature is valid.

13.9 CELLULAR AUTOMATA

A new and novel idea, studied by Guam [397], is the use of cellular automata in public-key cryptosystems. This system is still far too new and has not been studied extensively, but a preliminary examination suggests that it may have some cryptographic weaknesses [347]. Still, this is a promising area of research. Cellular automata have the property that, even if they are invertible, it is impossible to calculate the predecessor of an arbitrary state by reversing the rule for finding the successor. This sounds a lot like a trap-door one-way function.

13.10 ELLIPTIC CURVE CRYPTOSYSTEMS

Elliptic curves have been studied for many years, and there is an enormous amount of literature on the subject. In 1985, Neal Koblitz and V. S. Miller independently proposed using them for public-key cryptosystems [489,605]. They did not invent a cryptographic algorithm using elliptic curves, but they implemented existing public-key algorithms, like Diffie-Hellman, in elliptic curves over finite fields.

Elliptic curves are interesting because they provide a way of constructing "elements" and "rules of combining" that produce groups. These groups have enough familiar properties to build cryptographic algorithms, but they don't have certain properties that may facilitate cryptanalysis. For example, there is no good notion of "smooth." That is, there is no set of small elements in terms of which a random element has a good chance of being expressed by a simple algorithm. Hence, index calculus discrete logarithm algorithms do not work. See [605] for more details.

Elliptic curves over the finite field GF(2^n) are particularly interesting. The arithmetic processors for the underlying field are easy to construct and are relatively simple to implement for n in the range of 130 to 200. These systems have the potential to provide small and low-cost public-key cryptosystems. Many public-key algorithms, like Diffie-Hellman, ElGamal, and Schnorr, can be implemented in elliptic curves over finite fields.

The mathematics here are complex and beyond the scope of this book. Those interested in this topic are invited to read the two references mentioned above, and [65,585,491,95,506,493,507]. Next Computer, Inc.'s Fast Elliptic Encryption (FEE) uses elliptic curves [227a]. Koblitz also proposed public-key cryptosystems using hyperelliptic curves [490,492]. See also [811,676].

13.11 OTHER PUBLIC-KEY ALGORITHMS

Many other public-key algorithms have been proposed and broken over the years. The Matsumoto-Imai algorithm [569] was broken in [261]. The Cade algorithm was first proposed in 1985, broken in 1986 [451], and then strengthened in the same year [181]. In addition to these attacks, there are general attacks for decomposing polynomials over finite fields [368]. Any algorithm that gets its security from the composition of polynomials over a finite field should be looked upon with skepticism, if not outright suspicion.

The Yagisawa algorithm combines exponentiation mod p with arithmetic mod $p-1$ [892]; it was broken in [159]. Another public-key algorithm, proposed by Tsujii, Kurosawa, Itoh, Fujioka, and Matsumoto [856] is insecure [534]. A third system, Luccio-Mazzone [560], is insecure [424].

Gustavus Simmons suggested J-algebras as a basis for public-key algorithms [818,89]. This idea was abandoned after efficient methods for factorizing polynomials was invented [536]. Special polynomial semigroups have also been studied [891,545], but so far nothing has come of it. Harald Niederreiter proposed a public-key algorithm based on shift-register sequences [647].

Some European cryptographers have developed generalizations of RSA that use various permutation polynomials instead of exponentiation. Winfried Müller and Wilfried Nöbauer use Dickson polynomials [626,627,547]. Rudolph Lidl and Müller generalized this approach in [548,625] (a variant is called the Réidi scheme), and Nöbauer looked at its security in [657,658]. Peter Smith reinvented this scheme in 1993, calling it LUC [841,318]. These variants are all as secure as

RSA; there is really no security benefit to using any of them. Tatsuaki Okamoto and Kazuo Ohta compare a number of digital signature schemes in [675].

Prospects for creating radically new and different public-key cryptography algorithms seem dim. In 1988 Diffie noted that most public-key algorithms are based on one of three hard problems [295,297]:

> knapsack problem:
>
> Given a set of unique numbers, find a subset whose sum is N.
> discrete logarithm:
>
> If p is a prime and g and M are integers, find x such that $g^x = M \pmod{p}$
> factoring:
>
> If N is the product of two primes,
>
> a) factor N,
>
> b) given integers M and C, find d such that $M^d = C \pmod{N}$,
>
> c) given integers e and C, find M such that $M^e = C \pmod{N}$,
>
> d) given an integer x, decide whether there exists an integer y such that $x = y^2 \pmod{N}$.

According to Diffie [295,297], the discrete logarithm problem was suggested by J. Gill, the factoring problem by Knuth, and the knapsack problem by Diffie himself.

This narrowness in the mathematical foundations of public-key cryptography is worrisome. A breakthrough in either the problem of factoring or of calculating discrete logarithms could render whole classes of public-key algorithms insecure. Diffie points out [295,297] that this risk is mitigated by two factors:

> The operations on which public key cryptography currently depends—multiplying, exponentiating, and factoring—are all fundamental arithmetic phenomena. They have been the subject of intense mathematical scrutiny for centuries and the increased attention that has resulted from their use in public key cryptosystems has on balance enhanced rather than diminished our confidence.
>
> Our ability to carry out large arithmetic computations has grown steadily and now permits us to implement our systems with numbers sufficient in size to be vulnerable only to a dramatic breakthrough in factoring, logarithms, or root extraction.

As we have seen, not all public-key algorithms based on these problems are secure. The strength of any public-key algorithm depends more on the computational complexity of the problem upon which it is based; a computationally difficult problem does not necessarily imply a strong algorithm. Shamir listed three reasons why this is so [789]:

1. Complexity theory usually deals with single isolated instances of a problem. A cryptanalyst often has a large collection of statistically related problems to solve—several ciphertexts encrypted with the same key.
2. The computational complexity of a problem is typically measured by its worst-case or average-case behavior. To be useful as a cipher, the problem must be hard to solve in almost all cases.
3. An arbitrarily difficult problem cannot necessarily be transformed into a cryptosystem, and it must be possible to insert trapdoor information into the problem so that a shortcut solution is possible with this information and only with this information.

13.12 WHICH PUBLIC-KEY ALGORITHM IS BEST?

This question is muddied not only by the fact that different algorithms do different things, but also by patent and licensing issues. For encryption and digital signatures, RSA is the easiest to implement. ElGamal works well for encryption, and DSA is great for digital signatures. DSA is available royalty-free, making it the obvious choice. Diffie-Hellman is the easiest algorithm for key exchange. The various identification schemes also have their uses. I would suggest that before you use any other algorithm, do a literature search to make sure it hasn't been broken.

Example Implementations

It is one thing to design protocols and algorithms for key management, authentication, and encryption, but another thing entirely to implement them in operational systems. Often ideas that look good on paper don't work in real life. Maybe the bandwidth requirements are too large; maybe there is just too much delay in the protocol.

Most of the following key management systems have been implemented, at least in a test bed.

17.1 IBM SECRET-KEY MANAGEMENT PROTOCOL

This is a complete key management system for communications and file security on a computer network, using only symmetric cryptography. It was designed at IBM in the late 1970s by Ehrsam, Matyas, Meyer, and Tuchman [312,571].

This protocol has three main features: secure communications between a server and several terminals, secure file storage at the server, and secure communication among servers. The protocol doesn't really provide for direct terminal-to-terminal communication, although it can be modified to do that.

Each server on the network is attached to a cryptographic facility, which does all of the encrypting and decrypting. Each server has a **master key,** KM0, and two variants, KM1 and KM2. Both KM1 and KM2 are simple variants of KM0. These keys are used to encrypt other keys and to generate new keys. Each terminal has a **master terminal key,** KMT, which is used to facilitate key exchange among different computers.

The servers store KMT, encrypted with KM1. All other keys, such as those used to encrypt files of keys (called KNF), are stored in encrypted form under KM2.

The master key, KM0, is stored in some nonvolatile security module. Today it could be either a ROM key or a magnetic card, or it could be typed in manually by the user (probably as a text string and then key-crunched). KM1 and KM2 are not stored anywhere in the system, but they are computed from KM0 whenever they are needed. Session keys, for communication among servers, are generated with a pseudo-random process in the server. Keys to encrypt files for storage (KNF) are generated in the same manner.

The heart of the protocol is a tamper-resistant module, called a **cryptographic facility.** At both the server and the terminal, all encryption and decryption takes place within this facility. The most important keys, the keys used to generate the actual encryption keys, are also stored in this module. These keys can never be read once they are stored. Davies and Price [249] discuss this key management protocol in detail.

A Variation

A variation on this scheme of master and session keys can be found in [833]. This scheme is built around network nodes with key notarization facilities that serve local terminals. It is designed to provide:

- Secure two-way communication between any two terminal users.
- Secure communications using encrypted mail.
- Protection of personal files.
- Provision of a digital signature capability.

For communication and file transfer among users, the scheme uses keys generated in the key notarization facility and sent to the users encrypted under a master key. The identities of the users are incorporated with the key, to provide evidence that the session key has been used between a particular pair of users. This is the **key notarization** feature that is central to the system. Although the system does not use public-key cryptography, there is a digital-signature-like capability, in that the key could have only come from a particular source and could only be read at a particular destination.

17.2 MITRENET

One of the earliest implementations of public-key cryptography is the experimental system MEMO (MITRE Encrypted Mail Office). MITRE is a DoD contractor, a government think-tank. MEMO was a secure electronic-mail system for users in the MITRENET network. MEMO uses public-key cryptography for key exchange and DES for file encryption.

In the MEMO system, a Public Key Distribution Center is a separate node on the network. It contains all public keys, stored in EPROM, to prevent anyone from changing them. Private keys are generated by users or by the system.

Every time users log on to the network, they first establish a secure communications path with the Public Key Distribution Center. The users request a file of all the public keys from the Center. If the users pass an identification test using their individual private keys, the Center sends this list to each user's workstation. This list is encrypted using DES to ensure file integrity.

When users want to send a message to other users, they first generate a random DES key (actually, the system does this itself). Then they encrypt the file with the DES key and encrypt the DES key with the recipient's public key. Both the DES-encrypted file and the public-key-encrypted key are sent to the recipient.

MEMO makes no provision for lost keys. There is some provision for integrity checking of the messages, using checksums. There is no authentication built into the system.

The particular public-key implementation used for this system (Diffie-Hellman key exchange over $GF(2^{127})$) was proven insecure before the system was implemented (see Section 9.6), although it is easy to modify the system to use larger numbers. MEMO was intended mainly for experimental purposes and was never made operational on the real MITRENET system.

17.3 ISDN

Bell-Northern Research developed a prototype secure Integrated Services Digital Network (ISDN) Terminal [302,662,296,303]. The terminal can transmit and receive voice and data at 64 kbps. The terminal uses Diffie-Hellman key exchange, RSA digital signatures, and DES data encryption.

Keys

There is a long-term public/private key pair embedded in the phone. The private key is stored in a tamper-resistant area of the phone. The public key serves as the identification of the phone. These keys are part of the phone itself and cannot be altered in any way.

Additionally, there are two other public keys stored in the phone. One of these keys is the owner's public key. This key is used to authenticate commands from the owner. This key can be changed via a command signed by the owner. In this way one owner can transfer ownership of the phone to another person.

Another public key stored in the phone is the public key of the network. This key is used to authenticate commands from the network's key management facility. It is used to authenticate calls from other users on the network. This key can also be changed via a signed command from the owner. This permits the owners to move their phone from one network to another.

These keys are considered long-term keys: they are rarely, if ever, changed. There is also a short-term public/private key pair stored on the phone. These are encapsulated in a certificate signed by the key management facility. When two phones set up a call, they exchange certificates. The public key of the network is used to authenticate these certificates.

This exchange and verification of certificates only sets up a secure call from phone to phone. To set up a secure call from person to person, there is an additional piece of the protocol. The private key of the owner is stored on a hardware "ignition key," which is inserted into the telephone by the owner. This ignition key contains the owner's private key, encrypted under a secret password known only by the owner (not by the phone, not by the network's key management facility, not by anybody). It also contains a certificate signed by the network's key management facility that contains the owner's public key and some identifying information (name, company, security clearance, job title, favorite pizza toppings, etc.). This is also encrypted. To decrypt this information and enter it into the phone, the owners must type their secret password on the phone's touchpad. After the phone uses this information to set up calls, it is destroyed when the owners remove their ignition key.

The phone also stores a set of certificates from the network's key management facility. These certificates authorize particular users to use particular phones.

Calling

A call from Alice to Bob works as follows:

(1) Alice inserts her ignition key into the phone and enters her password.

(2) The phone interrogates the ignition key to determine Alice's identity.

(3) The phone checks its set of certificates to ensure that the user is authorized to use the particular phone.

(4) Alice dials the number; the phone places the call.

(5) The two telephones use a public-key cryptography key-exchange protocol to generate a unique and random session key. All subsequent protocol steps are encrypted using this key.

(6) The calling phone transmits its certificate and user authentication.

(7) The called phone authenticates the signatures on both the certificate and the user authentication using the network's public key.

(8) Bob's phone initiates a challenge-and-reply sequence. It demands real-time signed responses to time-dependent challenges. (This prevents an adversary from using certificates copied from a previous exchange.) One response must be signed by Alice's phone's private key; another must be signed by Alice's user's private key.

(9) Bob's phone rings.

(10) If he is home, he inserts his ignition key into the phone. His phone interrogates the ignition key and checks the certificate as in steps (2) and (3).

(11) Alice's phone transmits its certificate and user authentication.

(12) Alice's phone authenticates Bob's signatures as in step (7) and initiates a challenge-and-reply sequence as in step (8).

(13) Both phones display the identities of the other user and phone on their displays.

(14) The secure conversation begins.

(15) When one party hangs up, the session key is deleted, as are the certificates the called phone received from the calling phone and the certificates the calling phone received from the called phone.

Each DES key is unique to each call. It exists only inside the two phones for the duration of the call and is destroyed immediately afterward. If an adversary captures one of the phones involved in the call—or even both phones—the adversary will not be able to decrypt any of the previous calls between the two phones.

17.4 KERBEROS

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

The Kerberos authentication service was originally developed at MIT for Project Athena. The UNIX software has been put in the public domain and can be used by anybody. The Kerberos model is based in part on Needham and Schroeder's trusted third-party protocol [645] and on modifications by Denning and Sacco [272]. The original version of Kerberos, Version 4, is specified in [604,845]. (Versions 1 through 3 were internal development versions.) Version 5, modified from Version 4, is specified in [497,498]. Two good articles on using Kerberos in the real world are [456,457].

The Kerberos Model

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

Kerberos is a service Kerberos on the network. This is the trusted arbiter mentioned above. Kerberos keeps a database of clients and their secret keys. For a human user, the secret key is an encrypted password. Network services that require authentication, as well as clients who wish to use these services, register their secret key with Kerberos.

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

Kerberos provides three different levels of protection. (In mythology, Kerberos was a mythical three-headed guard dog.) It provides authentication at the beginning of a network connection, after which all further communications are assumed to come from the authenticated entity. It also provides authentication for each message sent from one entity to another. Finally, it provides both authentication and encryption for each message sent from one entity to another.

Software Modules

The implementation of Kerberos generally available contains several software modules. These are:

Kerberos applications library provides an interface for application clients and servers. It contains routines for creating and reading authentication requests and routines for creating authenticated and encrypted messages.

Encryption library: a DES implementation is provided with Kerberos (users outside the United States may find this removed when the code is exported, but they can easily add it back in). The DES included with the source code may be replaced with other DES implementations or a different encryption routine. Kerberos Version 4 provided a nonstandard **Propagating Cipher Block Chaining (PCBC)** mode for authentication. This mode is insecure (see Section 8.1.7). Kerberos Version 5 uses CBC mode.

Database library: Kerberos has simple database needs—a record for each entity containing its name, secret key, expiration date of the secret key, and some administrative information. Since the security of the whole system depends on the security of this database, it must run on a secure machine.

Database administration programs administer the Kerberos database.

Administration server provides a read-write network interface to the database. It is used to add, delete, and update entries in the database. For integrity reasons, the server side of this program must run on the machine housing the Kerberos database. Of course, this machine must be secure.

Authentication server provides a read-only network interface to the database. It is used to authenticate entities and to generate session keys. Since this server does not modify the database, it can run on machines with a read-only copy of the Kerberos database. These machines must also be secure.

Database propagation software. This manages replication of the Kerberos database. To keep Kerberos from collapsing under all the requests for authentication, it is often desirable to have multiple authentication servers on multiple machines. Each of these servers receives a copy of the Kerberos database at regular intervals.

User programs. These are programs for logging into Kerberos, changing a Kerberos password, and displaying and destroying authentication information.

Applications.

How Kerberos Works

This section discusses Kerberos Version 5. I will outline the differences between Version 4 and Version 5 in a later section. The Kerberos protocol is straightforward (see Figure 17.1). A client requests a ticket for a **Ticket-Granting Service (TGS ticket)** from the Kerberos service. This ticket is sent to the client, encrypted in the client's secret key. To use a particular service, the client requests a ticket for that service from the **Ticket-Granting Server (TGS).** Assuming everything is in order, the TGS sends the ticket back to the client. The client then presents this ticket to the server along with an authenticator. Again, if there's nothing wrong with the client's credentials, the server lets the client have access to the service.

Credentials

There are two types of credentials used in Kerberos: **tickets** and **authenticators.** (The rest of this section uses the notation used in Kerberos documents—see Table 17.1.) A ticket is used to pass securely the identity of the client for whom the ticket was issued to the server. It also contains information that the server can use to ensure that the client using the ticket is the same client to which the ticket was issued. An authenticator is an additional credential, presented with the ticket.

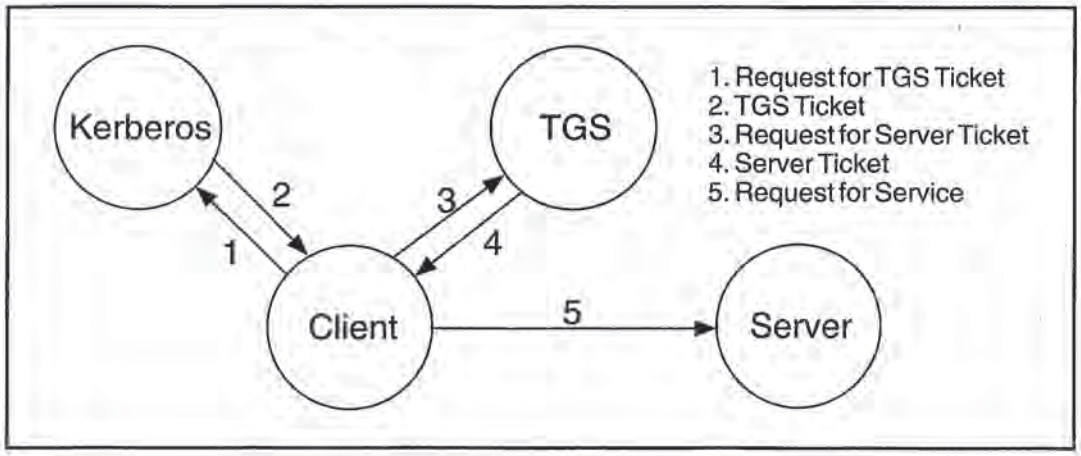


Figure 17.1 Kerberos authentication protocols.

TABLE 17.1
Kerberos Table of Abbreviations

c	= client
s	= server
addr	= client's network address
times	= beginning and ending validity time for a ticket
Kerberos	= authentication server
K_x	= x's secret key
$K_{x,y}$	= session key for x and y
$\{m\}K_x$	= m encrypted in x's secret key
$T_{x,y}$	= x's ticket to use y
$A_{x,y}$	= authenticator from x to y

A Kerberos ticket looks like this:

$$T_{c,s} = s, \{c, \text{addr}, \text{times}, K_{c,s}\}K_s$$

A ticket is good for a single server and a single client. It contains the name of the client, the name of the server, the network address of the client, a timestamp, and a session key. This information is encrypted with the server's secret key. Once the client gets this ticket, she can use it multiple times to gain access to the server, until the ticket expires. The client cannot decrypt the ticket (she does not know the server's secret key), but she can present it to the server in its encrypted form. No one listening on the network can read or modify the ticket as it passes through the network.

A Kerberos authenticator looks like:

$$A_{c,s} = \{c, \text{time}, \text{key}\}K_{c,s}$$

The client generates it every time she wishes to use a service on the server. It contains the name of the client, a timestamp, an optional additional session key, all encrypted with the session key shared between the client and the server. Unlike a ticket, it can only be used once. However, since the client can generate authenticators as needed (it knows the shared secret key), this is not a problem.

The authenticator serves two purposes. First, it contains some plaintext encrypted in the session key. This proves that the sender of the authenticator also knows the key. Just as important, the sealed plaintext includes the time of day; an eavesdropper who records both the ticket and the authenticator can't replay them two days later.

Getting an Initial Ticket

The client has one piece of information that can prove her identity: her password. Obviously we don't want the client to send this password over the network. The Kerberos protocol minimizes the chance that this password will be compromised, while at the same time not allowing a user to properly authenticate herself unless she knows the password.

The client sends a message to the Kerberos authentication server, containing the user's name and the name of her TGS server. (There can be many TGS servers.) In reality, the user probably just enters her username onto the system, and the log-in program sends the request.

The Kerberos authentication server looks up the client in his database. If the client is in the database, Kerberos generates a session key to be used between the client and the TGS. This is called a **Ticket Granting Ticket,** or a **TGT.** It encrypts that session key with the client's secret key. Then it creates a TGT for the client to authenticate herself to the TGS and encrypts that in the TGS's secret key. The authentication server sends both of these encrypted messages back to the client. Figure 17.2 shows this exchange.

The client now decrypts the first message and retrieves the session key. The secret key is a one-way hash of the user's password, so a legitimate user will have no trouble doing this. If the user were an imposter, she would not know the correct password and therefore could not decrypt the response from the Kerberos authentication server. Access would be denied, and she wouldn't be able to get the ticket or the session key.

The client saves the TGT and session key and erases the password and the one-way hash. This information is erased to reduce the chance of compromise. If an adversary manages to copy the client's memory, the result will only be the TGT and the session key. These are valuable pieces of information, but only during the lifetime of the TGT. After the TGT expires, they will be worthless.

If everyone is who they say they are, the client can now prove her identity to the TGS for the lifetime of the TGT.

Getting Server Tickets

A client has to obtain a separate ticket for each service she wants to use. The TGS grants tickets for individual servers.

When a client requires a ticket that she does not already have, she sends a request to the TGS. (In reality, the program would do this automatically, and it would be invisible to the user.) Figure 17.3 shows the exchange.

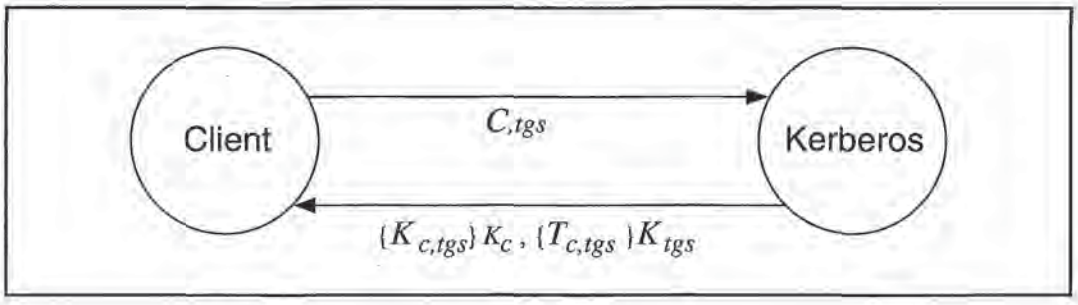


Figure 17.2
Getting a TGS ticket.

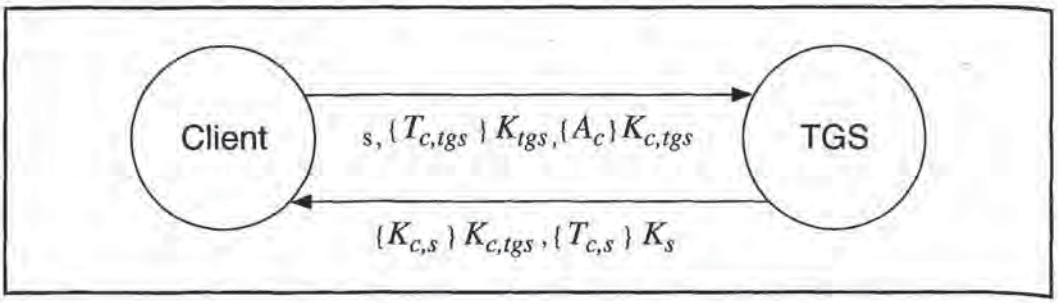


Figure 17.3
Getting a server ticket.

First, the client creates an authenticator. The authenticator consists of the client's name, the client's address, and a timestamp, encrypted with the session key generated by the Kerberos authentication server. The request consists of the name of the server, the TGT received from Kerberos (already encrypted with the TGS's secret key), and the encrypted authenticator.

The TGS, upon receiving the request, decrypts the TGT with his secret key. Then he uses the session key included in the TGT to decrypt the authenticator. Finally, he compares the information in the authenticator with the information in the ticket, the network address of the client with the address the request was sent from, and the timestamp with the current time. If everything matches, he allows the request to proceed.

Checking timestamps assumes that all machines have synchronized clocks, at least to within several minutes. If the time in the request is too far in the future or the past, the TGS treats the request as an attempt to replay a previous request. The TGS should also keep track of all live authenticators, i.e., past requests with timestamps that are still valid. Another request received with the same ticket and timestamp as one already received can be ignored.

The TGS responds to a valid request by returning a valid ticket for the client to present to the server. The ticket contains the name of the client, the network address of the client, a timestamp, an expiration time for the ticket, and the session key it just created, all encrypted with the server's secret key, and the name of the server. The TGS also creates a new session key for the client and the server, encrypted with the session key shared by the client and the TGS. Both of these messages are then sent back to the client.

The client decrypts the message and extracts the session key.

Requesting a Service

Now the client is ready to authenticate herself to the server. She creates a message very similar to the one sent to the TGS (which makes sense, since the TGS is a service). Figure 17.4 shows this.

The client creates an authenticator, consisting of the client's name, the client's network address, and a timestamp, encrypted with the session key for the client and the server that the TGS generated. The request consists of the ticket received from Kerberos (already encrypted with the server's secret key) and the encrypted authenticator.

The server decrypts and checks the ticket and the authenticator, as discussed previously, and also checks the client's address and the timestamp. If everything

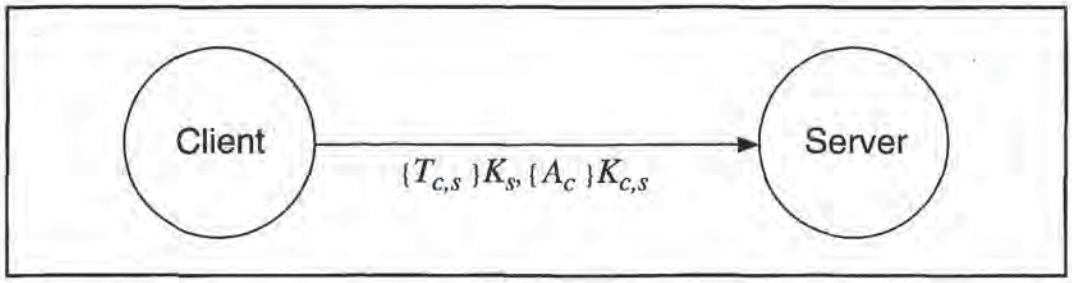


Figure 17.4
Requesting a service.

checks out, the server knows that, according to Kerberos, the client is who she says she is.

For applications that require mutual authentication, the server sends the client back a message consisting of the timestamp plus one, encrypted with the session key (see Figure 17.5). This proves that the server knew his secret key and could decrypt the ticket and therefore the authenticator.

The client and the server can encrypt future messages with the shared key, if desired. Since only the client and the server share this key, they both can assume that a recent message encrypted in that key originated with the other party.

Kerberos Version 4

The previous sections discussed Kerberos Version 5. Kerberos Version 4 differed slightly in the messages and the construction of the tickets and authenticators.

In Kerberos Version 4, the five messages looked like:

1. Client to Kerberos: c, tgs
2. Kerberos to client: $\{K_{c,tgs}, \{T_{c,tgs}\}K_{tgs}\}K_c$
3. Client to TGS: $\{A_c\}K_{c,tgs}, \{T_{c,tgs}\}K_{tgs}, s$
4. TGS to client: $\{K_{c,s}, \{T_{c,s}\}K_s\}K_{c,tgs}$
5. Client to server: $\{A_c\}K_{c,s}, \{T_{c,s}\}K_s$

$T_{c,s} = \{s,c,\text{addr},\text{time},\text{life},K_{c,s}\}K_s$

$A_{c,s} = \{c,\text{addr},\text{time}\}K_{c,s}$

Messages 1, 3, and 5 are identical. The double encryption of the ticket in steps 2 and 4 has been removed in Version 5. The Version 5 ticket adds the ability to have

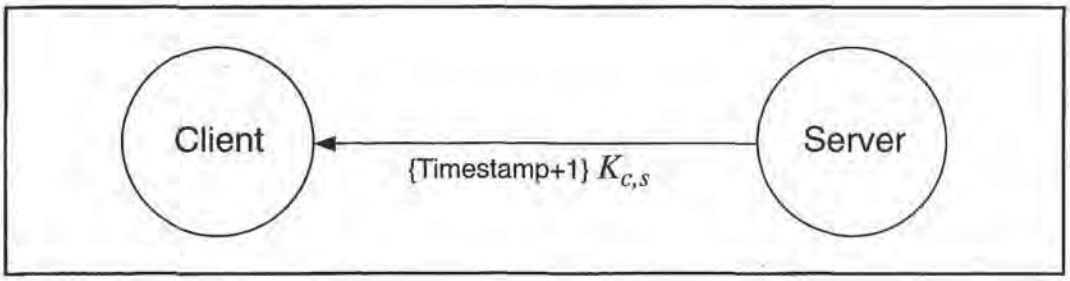


Figure 17.5
Mutual authentication.

multiple addresses, and it replaces a "lifetime" field with a beginning and ending time. The Version 5 authenticator adds the option of including an additional key.

Security of Kerberos

Steve Bellovin and Michael Merritt, in their paper "Limitations of the Kerberos Protocol" [58], discuss several potential security vulnerabilities of Kerberos. Although this paper was written about the Version 4 protocols, many of their comments also apply to Version 5.

It may be possible to cache and replay old authenticators. Although timestamps are supposed to prevent this, replays can be done during the lifetime of the ticket. Servers are supposed to store all valid tickets to prevent replays, but this is not always possible.

Authenticators rely on the fact that all the clocks in the network are more or less synchronized. If a host can be fooled about the correct time, then an old authenticator can be replayed without any problem. Most network time protocols are insecure, so this can be a serious problem.

Kerberos is also vulnerable to password-guessing attacks. An intruder can collect tickets and then try to decrypt them. Remember that the average user doesn't usually choose good passwords. If Mallet collects enough tickets, his chances of recovering a password are good.

Perhaps the most serious attack involves malicious software. The Kerberos protocols rely on the fact that the Kerberos software is trustworthy. There's nothing to stop Mallet surreptitiously replacing all client Kerberos software with a version that, in addition to completing the Kerberos protocols, records passwords. This is a problem with any cryptographic software package on an insecure computer, but the widespread use of Kerberos in these environments makes it a particularly tempting target.

Future of Kerberos

In [497], Kohl discusses items that may be incorporated into future versions of Kerberos:

> Public-key cryptosystems. The current Kerberos protocols are designed for symmetric algorithms. There is some work towards incorporating public-key algorithms into Kerberos, but public-key algorithms have very different characteristics. Taking advantage of public-key cryptography would require a complete reworking of the protocol.
>
> Smart Cards. These are hand-held devices which would augment normal password security methods.
>
> Remote Administration. This feature would provide a standard interface to the Kerberos database for system administrators.
>
> Directional inter-realm keys. This would increase security across different networks.
>
> Database propagations. There is thought of turning the Kerberos database into a distributed database.

Validation suites. There is currently no complete validation suite to verify that the protocol is properly implemented. Such a suite could prevent security problems resulting from faulty implementation of the protocols, and would help insure interoperability between implementations.

Applications. There are many applications that would benefit from the addition of Kerberos authentication (this is known as "Kerberizing" an application).

Licenses

Kerberos is public domain, and free source code is available from MIT's Project Athena. Actually implementing this code into a working environment is another story. The Open Computer Security Group sells and supports a commercial version of the Kerberos Version 5 protocols for SUN, RS/6000, NeXT, HP 9000, Sequent, Pyramid, NCR, IBM mainframes, VMS, MS-DOS, and Macintosh. Interested parties should contact:

Gene Egan
Open Computing Security Group
2451 152nd Avenue NE
Redmond, WA 98052
Tel: (206) 883-8721

17.5 KRYPTOKNIGHT

KryptoKnight (Kryptonite—get it?) is an authentication and key distribution system designed by IBM. It is a secret-key protocol and uses either DES in CBC mode (see Section 8.1.3) or a modified version MD5 (see Section 14.5).

KryptoKnight supports four security services:

- User authentication (called Single Sign-On)
- Two-party authentication
- Key distribution
- Authentication of origin and content of data

From a user's perspective, KryptoKnight is similar to Kerberos. For further information on KryptoKnight and its protocols, read [615,108,109,110].

17.6 ISO AUTHENTICATION FRAMEWORK

Public-key cryptography has been recommended for use with the ISO authentication framework, also known as the X.509 protocols [188]. This framework provides for authentication across networks. Although no particular algorithms have been specified for either security or authentication, the specification calls for an algorithm that can be used for both security and authentication, and one in which the encryption and decryption processes are

inverses of each other. (This precludes ElGamal and DSA and strongly implies the RSA algorithm.) There are provisions, however, for multiple algorithms and hash functions.

Certificates

The framework is certificate-based. Each user has a distinct name. A trusted certification authority assigns a unique name to each user and issues a certificate containing the name and the user's public key.

An X.509 certificate looks like [188]:

```
certificate ::= SIGNED SEQUENCE  (
  signature   AlgorithmIdentifier,
  issuer   Name,
  validity   Validity ::= SEQUENCE (
    notBefore   UTCTime,
    notAfter   UTCTime)
  subject   Name,
  subjectPublicKeyInfo   SubjectPublicKeyInfo ::= SEQUENCE  (
    algorithm   AlgorithmIdentifier,
    subjectPublicKey   BIT STRING))
```

A Certifying Authority (CA) signs all certificates. If Alice and Bob want to communicate, each of them has to verify the signature of the other person's certificate. If they use the same CA, this is easy. If they use different CAs, this is more complicated. Think of a tree structure, with different CAs certifying other CAs and users. On the top there is one master CA. Each CA stores the certificate obtained from its superior CA, as well as all the certificates issued by it. Alice and Bob have to traverse the certification tree, looking for a common trusted point.

Figure 17.6 illustrates this. Alice's certificate is certified by CA_A; Bob's is certified by CA_B. CA_A's certificate (needed to verify CA_A's signature on Alice's certificate) is certified by CA_C. CA_C's certificate is certified by CA_E; CA_E's certificate is certified by CA_F. CA_B's certificate is certified by CA_D; CA_D's certificate is certified by CA_F. This is a common trusted point and one that can certify Alice to Bob and Bob to Alice.

This procedure becomes complicated if Alice and Bob have to search the network for the different certificates, but in reality they can both cache all the certificates to speed up the process.

Certificates have a specific validity period. When a certificate expires, it should be removed from any public directories maintained by the CAs. The issuing CA, however, should maintain a copy of the certificate. It will be required to resolve any disputes that might arise.

The Protocols

Alice wants to communicate with Bob. First she goes to a directory and obtains what is called a "certification path" from Alice to Bob, and Bob's public key. At

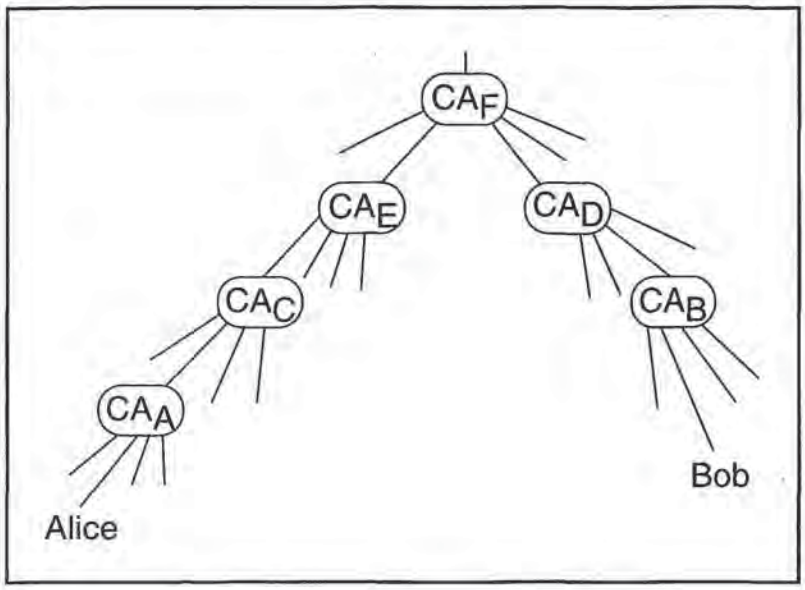


Figure 17.6
Sample certification hierarchy.

this point Alice can initiate either a one-way, two-way, or three-way authentication protocol.

The one-way protocol is a single communication from Alice to Bob. It establishes the identities of both Alice and Bob and the integrity of any information communicated by Alice to Bob. It also prevents any replay attacks in the communication.

The two-way protocol is identical to the one-way protocol, but it also adds a reply from Bob. It establishes that Bob, and not an imposter, sent the reply. It also establishes the secrecy of both communications and prevents replay attacks.

Both the one-way and two-way protocols use timestamps. A three-way protocol adds another message from Alice to Bob and obviates the need for timestamps.

The one-way protocol is:

(1) Alice generates a random number, R_A.

(2) Alice constructs a message, $M = (T_A, R_A, I_B, \text{data})$, in which T_A is Alice's timestamp, I_B is Bob's identity, and data is an arbitrary piece of information. The data may be encrypted with Bob's public key, E_B, for security.

(3) Alice sends $(C_A, D_A(M))$ to Bob.

(4) Bob verifies C_A and obtains E_A. He makes sure these keys have not expired.

(5) Bob uses E_A to decrypt $D_A(M)$. This verifies both Alice's signature and the integrity of the signed information.

(6) Bob checks the I_B in M for accuracy.

(7) Bob checks the T_A in M and confirms that the message is current.

(8) As an option, Bob can check R_A in M against a database of old random numbers to ensure the message is not an old one being replayed.

The two-way protocol consists of the one-way protocol and then a similar one-way protocol from Bob to Alice. After executing steps (1) through (8) of the one-way protocol, the two-way protocol continues with:

(9) Bob generates another random number, R_B.

(10) Bob constructs a message, $M' = (T_B, R_B, I_A, R_A, \text{data})$, in which T_B is Bob's timestamp, I_A is Alice's identity, and data is arbitrary. The data may be encrypted with Alice's public key, E_A, for security. R_A is the random number Alice generated in step (1).

(11) Bob sends $D_B(M')$ to Alice.

(12) Alice uses E_B to decrypt $D_B(M')$. This verifies both Bob's signature and the integrity of the signed information.

(13) Alice checks the I_A in M' for accuracy.

(14) Alice checks the T_B in M' and confirms that the message is current.

(15) As an option, Alice can check the R_B in M' to ensure the message is not an old one being replayed.

The three-way protocol accomplishes the same thing as the two-way protocol, but without timestamps. Steps (1) through (15) are identical to the two-way protocol, with $T_A = T_B = 0$.

(16) Alice checks the received version of R_A against the R_A she sent to Bob in step (3).

(17) Alice sends $D_A(R_B)$ to Bob.

(18) Bob uses E_A to decrypt $D_A(R_B)$. This verifies both Alice's signature and the integrity of the signed information.

(19) Bob checks the received version of R_B against the R_B he sent to Alice in step (10).

17.7 PRIVACY-ENHANCED MAIL (PEM)

PEM is the Internet Privacy-Enhanced Mail standard, adapted by the Internet Architecture Board (IAB) to provide secure electronic mail over the Internet. It was initially designed by the Internet Resources Task Force (IRTF) Privacy and Security Research Group (PSRG), and then handed over to the Internet Engineering Task Force (IETF) PEM Research Group. The PEM protocols provide for encryption, authentication, message integrity, and key management.

The complete PEM protocols were initially detailed in a series of RFCs (Requests for Comment) in [551] and then revised in [552]. The third iteration of the protocols [553,477,554] are summarized in a paper by Matthew Bishop [112,113], a member of the PSRG. The protocols were modified and improved, and the final protocols are detailed in another series of RFCs [555,476,39,468]. Another paper by Matthew Bishop [114] details the changes.

PEM is an inclusive standard. The PEM procedures and protocols are intended to be compatible with a wide range of key-management approaches, including both secret-key and public-key approaches for encryption of data encrypting keys. Symmetric cryptography is used for message-text encryption. Cryptographic hash algorithms are used for message integrity. Other documents specify supporting key-management mechanisms based on the use of public-key certificates; algorithms, modes, and associated identifiers; and details of paper and electronic formats and procedures for the key-management infrastructure being established in support of these services.

Initially, PEM supports only certain algorithms. Messages are encrypted with DES in CBC mode. Authentication, provided by something called a **Message Integrity Check (MIC),** uses either MD2 or MD5. Symmetric key management can use either DES in ECB mode or triple-DES, using two keys (called EDE mode). PEM also supports the use of public-key certificates for key management, using the RSA algorithm (key length up to 1024 bits) and the X.509 standard for certificate structure.

End-to-end cryptography provides three "privacy-enhancement services": confidentiality, authentication, and message integrity. There are no special processing requirements imposed on the electronic mail system. PEM can be incorporated selectively, by site or by user, without impacting the rest of the network.

PEM Documents

There are four PEM documents:

RFC 1421, Part I, Message Encryption and Authentication Procedures: This document defines message encryption and authentication procedures in order to provide privacy-enhanced mail services for electronic mail transfer in the Internet.

RFC 1422, Part II, Certificate-Based Key Management: This document defines a supporting key management architecture and infrastructure, based on public-key certificate techniques to provide keying information to message originators and recipients.

RFC 1423, Part III, Algorithms, Modes, and Identifiers: This document provides definitions, formats, references, and citations for cryptographic algorithms, usage modes, and associated identifiers and parameters.

RFC 1424, Part IV, Key Certification and Related Services: This document describes three types of service in support of PEM: key certification, certificate revocation list (CRL) storage, and CRL retrieval.

Certificates

PEM is compatible with the authentication framework described in [188]. PEM is a superset of X.509; it establishes procedures and conventions for a key management infrastructure for use with PEM and with other protocols, from both the TCP/IP and OSI suites, in the future.

The key-management infrastructure establishes a single root for all certification within the Internet: the Internet Policy Registration Authority (IPRA). The IPRA establishes global policies that apply to all certification effected under this hierarchy. Beneath the IPRA root are Policy Certification Authorities (PCAs), each of which establishes and publishes its policies for registration of users or organizations. Each PCA is certified by the IPRA. Below PCAs, Certification Authorities (CAs) are established to certify users and subordinate organizational entities (e.g., departments, offices, subsidiaries, etc.). Initially, the majority of users are expected to be registered with some organization.

Some PCAs are expected to provide certification for users who wish to register independent of any organization. For users who wish anonymity while taking advantage of PEM privacy facilities, one or more PCAs are expected to be established with policies that allow for registration of users, under subordinate PCAs, who do not wish to disclose their identities.

PEM Messages

The heart of PEM is its message format. Figure 17.7 shows an encrypted message using symmetric key management, Figure 17.8 shows an encrypted message using public-key key management, and Figure 17.9 shows an authenticated (but unencrypted) message using public-key key management.

```
—-BEGIN PRIVACY-ENHANCED MESSAGE—-
Proc-Type: 4,ENCRYPTED
Content-Domain: RFC822
DEK-Info: DES-CBC,F8143EDE5960C597
Originator-ID-Symmetric: linn@zendia.enet.dec.com,,
Recipient-ID-Symmetric: linn@zendia.enet.dec.com,ptf-kmc,3
Key-Info: DES-ECB,RSA-MD2,9FD3AAD2F2691B9A,B70665BB9BF7CBCDA60195DB94F727D3
Recipient-ID-Symmetric: pem-dev@tis.com,ptf-kmc,4
Key-Info: DES-ECB,RSA-MD2,161A3F75DC82EF26,E2EF532C65CBCFF79F83A2658132DB47

LLrHB0eJzyhP+/fSStdW8okeEnv47jxe7SJ/iN72ohNcUk2jHEUSoH1nvNSIWL9M
8tEjmF/zxB+bATMtPjCUWbz8Lr9wloXIkjHUlBLpvXR0UrUzYbkNpk0agV2IzUpk
J6UiRRGcDSvzrsoK+oNvqu6z7Xs5Xfz5rDqUcMlK1Z6720dcBWGGsDLpTpSCnpot
dXd/H5LMDWnonNvPCwQUHt==
—-END PRIVACY-ENHANCED MESSAGE—-
```

Figure 17.7
Example encapsulated message (symmetric case).

```
—-BEGIN PRIVACY-ENHANCED MESSAGE—-
Proc-Type: 4,ENCRYPTED
Content-Domain: RFC822
DEK-Info: DES-CBC,BFF968AA74691AC1
Originator-Certificate:
 MIIB1TCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDzAN
 BgNVBAsTBk5PVEFSWTAeFw05MTA5MDQxODM4MTdaFw05MzA5MDMxODM4MTZaMEUx
 CzAJBgNVBAYTA1VTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5jLjEU
 MBIGA1UEAxMLVGVzdCBVc2VyIDEwWTAKBgRVCAEBAgICAANLADBIAkEAwHZH17i+
 yJcqDtjJCowzTdBJrdAiLAnSC+CnnjOJELyuQiBgkGrgIh3j8/x0fM+YrsyF1u3F
 LZPVtzlndhYFJQIDAQABMA0GCSqGSIb3DQEBAgUAA1kACKrOPqphJYw1j+YPtcIq
 iWlFPuN5jJ79Khfg7ASFxskYkEMjRNZV/HZDZQEhtVaU7Jxfzs2wfX5byMp2X3U/
 5XUXGx7qusDgHQGs7Jk9W8CW1fuSWUgN4w==
Key-Info: RSA,
 I3rRIGXUGWAF8js5wCzRTkdhO34PTHdRZY9TuvmO3M+NM7fx6qc5udixps2LngO+
 wGrtiUm/ovtKdinz6ZQ/aQ==
Issuer-Certificate:
 MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDTAL
 BgNVBAsTBFRMQ0EwHhcNOTEwOTAxMDgwMDAwWhcNOTIwOTAxMDc1OTU5WjBRMQsw
 CQYDVQQGEwJVUzEgMB4GA1UEChMXU1NBIERhdGEgU2VjdXJpdHksIEluYy4xDzAN
 BgNVBAsTBkJldGEgMTEPMA0GA1UECxMGTk9UQVJZMHAwCgYEVQgBAQICArwDYgAw
 XwJYCsnp6lQCxYykNlODwutF/jMJ3kL+3PjYyHOwk+/9rLg6X65B/LD4bJHtO5XW
 cqAz/7R7XhjYCmOPcqbdzoACZtIlETrKrcJiDYoP+DkZ8k1gCk7hQHpbIwIDAQAB
 MA0GCSqGSIb3DQEBAgUAA38AAICPv4f9Gx/tY4+p+4DB7MV+tKZnvBoy8zgoMGOx
 dD2jMZ/3HsyWKWgSFOeH/AJB3qr9zosG47pyMnTf3aSy2nBO7CMxpUWRBcXUpE+x
 EREZd9++32ofGBIXaialnOgVUnOOzSYgugiQO77nJLDUj0hQehCizEs5wUJ35a5h
MIC-Info: RSA-MD5,RSA,
 UdFJR8u/TIGhfH65ieewe2lOW4tooa3vZCvVNGBZirf/7nrgzWDABz8w9NsXSexv
 AjRFbHoNPzBuxwmOAFeAOHJszL4yBvhG
Recipient-ID-Asymmetric:
 MFExCzAJBgNVBAYTA1VTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5j
 LjEPMA0GA1UECxMGQmVOYSAxMQ8wDQYDVQQLEwZOT1RBUlk=,
 66
Key-Info: RSA,
 O6BS1ww9CTyHPtS3bMLD+LOhejdvX6Qv1HK2ds2sQPEaXhX8EhvVphHYTjwekdWv
 7xOZ3Jx2vTAhOYHMcqqCjA==

qeWlj/YJ2Uf5ng9yznPbtDOmYloSwIuV9FRYx+gzY+8iXd/NQrXHfi6/MhPfPF3d
jIqCJAxvld2xgqQimUzoS1a4r7kQQ5c/Iua4LqKeq3ciFzEv/MbZhA==
—-END PRIVACY-ENHANCED MESSAGE—-
```

Figure 17.8
Example encapsulated message (asymmetric case).

```
 ——-BEGIN PRIVACY-ENHANCED MESSAGE——-
Proc-Type: 4,MIC-ONLY
Content-Domain: RFC822
Originator-Certificate:
 MIIBlTCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDzAN
 BgNVBAsTBk5PVEFSWTAeFw05MTA5MDQxODM4MTdaFw05MzA5MDMxODM4MTZaMEUx
 CzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cmlOeSwgSW5jLjEU
 MBIGA1UEAxMLVGVzdCBVc2VyIDEwWTAKBgRVCAEBAgICAANLADBIAkEAwHZHl7i+
 yJcqDtjJCowzTdBJrdAiLAnSC+CnnjOJELyuQiBgkGrgIh3j8/x0fM+YrsyF1u3F
 LZPVtzlndhYFJQIDAQABMA0GCSqGSIb3DQEBAgUAA1kACKrOPqphJYw1j+YPtcIq
 iWlFPuN5jJ79Khfg7ASFxskYkEMjRNZV/HZDZQEhtVaU7Jxfzs2wfX5byMp2X3U/
 5XUXGx7qusDgHQGs7Jk9W8CW1fuSWUgN4w==
Issuer-Certificate:
 MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDTAL
 BgNVBAsTBFRMQ0EwHhcNOTEwOTAxMDgwMDAwWhcNOTIwOTAxMDc1OTU5WjBRMQsw
 CQYDVQQGEwJVUzEgMB4GA1UEChMXUlNBIERhdGEgU2VjdXJpdHksIEluYy4xDzAN
 BgNVBAsTBkJldGEgMTEPMA0GA1UECxMGTk9UQVJZMHAwCgYEVQgBAQICArwDYgAw
 XwJYCsnp6lQCxYykNlODwutF/jMJ3kL+3PjYyHOwk+/9rLg6X65B/LD4bJHtO5XW
 cqAz/7R7XhjYCmOPcqbdzoACZtIlETrKrcJiDYoP+DkZ8k1gCk7hQHpbIwIDAQAB
 MA0GCSqGSIb3DQEBAgUAA38AAICPv4f9Gx/tY4+p+4DB7MV+tKZnvBoy8zgoMGOx
 dD2jMZ/3HsyWKWgSFOeH/AJB3qr9zosG47pyMnTf3aSy2nBO7CMxpUWRBcXUpE+x
 EREZd9++32ofGBIXaialnOgVUnOOzSYgugiQ077nJLDUj0hQehCizEs5wUJ35a5h
MIC-Info: RSA-MD5,RSA,
 jV2OfH+nnXHU8bnL8kPAad/mSQlTDZlbVuxvZAOVRZ5q5+Ejl5bQvqNeqOUNQjr6
 EtE7K2QDeVMCyXsdJlA8fA==

LSBBIG1lc3NhZ2UgZm9yIHVzZSBpbiBOZXN0aW5nLgOKLSBGb2xsb3dpbmcgaXMg
YSBibGFuayBsaW5lOgOKDQpUaGlzIGlzIHRoZSBlbmQuDQo=
——-END PRIVACY-ENHANCED MESSAGE——-
```

Figure 17.9
Example encapsulated MIC-ONLY message (asymmetric case).

The first field is "Proc-Type," and identifies the type of processing performed on the message. There are three types of messages. The "ENCRYPTED" specifier signifies that the message is encrypted and authenticated. The "MIC-ONLY" and "MIC-CLEAR" specifier indicates that the message is signed, but not encrypted. MIC-CLEAR messages are not encoded, and can be read using non-PEM software. MIC-ONLY messages must be transformed to a human-readable form before being

read, so PEM software is needed. A PEM message is always signed; it is optionally encrypted.

The next field, "Content-Domain," has nothing to do with security. It specifies the type of mail message.

The "DEK-Info" field specifies information pertaining to the **Data Exchange Key (DEK),** the encryption algorithm used to encrypt the text and any parameters associated with the encryption algorithm. Only one algorithm is currently specified: DES in CBC mode. This is specified by "DES-CBC." The second field specifies the Initialization Vector, used by DES in CBC mode. Other algorithms may be specified by PEM in the future; their use will be noted in the DEK-Info and in other fields that identify algorithms.

For messages with symmetric key management, the next field is "Originator-ID-Symmetric." There are three subfields. The first subfield identifies the sender by a unique electronic mail address. The second subfield is optional and identifies the authority that issued the interchange key. The third subfield is an optional Version/Expiration subfield.

Continuing with the symmetric key-management case, the next field deals with the recipients. There are two fields for each recipient: "Recipient-ID-Symmetric" and "Key-Info." The "Recipient-ID-Symmetric" field has three subfields; it identifies the receiver in the same way that "Originator-ID-Symmetric identified the sender.

The "Key-Info" field specifies the key-management parameters. This field has four subfields. The first subfield specifies the algorithm used to encrypt the DEK. Since the key management in this message is symmetric, the sender and receiver have to share a common key. This is called the **Interchange Key (IK),** which is used to encrypt the DEK. The DEK can be either encrypted using DES in Electronic Codebook Mode (denoted by "DES-ECB") or triple-DES (denoted by "DES-EDE") for DES-Encrypt Decrypt Encrypt. The second subfield specifies the MIC algorithm. It can be either MD2 (denoted by "RSA-MD2") or MD5 (denoted by "RSA-MD5"). The third subfield is the DEK, encrypted with the IK. The fourth field is the MIC, also encrypted with the IK.

For messages with public-key (asymmetric) key management, the headers are different. After the "DEK-Info" field comes the "Originator-Certificate" field. The certificate follows the X.509 standard (see Section 17.6). The next field is "Key-Info." There are two subfields. The first subfield specifies the public-key algorithm used to encrypt the DEK; currently only RSA is supported. The next subfield is the DEK, encrypted in the originator's public key. This is an optional field, intended to permit the originator to decrypt an individual message in the event that it is returned by the mail system. The next field "Issuer-Certificate," is the certificate of whomever signed the Originator-Certificate.

Continuing with the asymmetric key-management case, the next field is "MIC-Info." The first subfield specifies the algorithm under which the MIC was computed. The second subfield specifies the algorithm under which the MIC was signed. The third subfield consists of the MIC, signed by the sender's private key.

Still continuing with asymmetric key management, the next fields deal with the recipients. There are two fields for each recipient: "Recipient-ID-Asymmetric" and "Key-Info." The "Recipient-ID-Asymmetric" field has two subfields. The first identifies the authority that issued the receiver's public key; the second is an optional Version/Expiration subfield. The "Key-Info" field specifies the key-management parameters. The first subfield identifies the algorithm used to encrypt the message; the second subfield is the DEK, encrypted with the receiver's public key.

Security of PEM

RSA keys in PEM can range from 508 bits to 1024 bits. This should be long enough for anyone's security needs. A more likely attack would be against the key-management protocols. Mallet could steal your private key—it should never be written down anywhere—or attempt to fool you into accepting a bogus public key. The key certification provisions of PEM make this unlikely if everyone follows proper procedures, but people have been known to be sloppy.

A more insidious attack would be for Mallet to modify the implementation of PEM running on your system. The modified implementation could surreptitiously send Mallet all of your mail, encrypted with his public key. It could even send him a copy of your private key. If the modified implementation works well, you will never know what is happening.

There's no real way to prevent this kind of attack. You could use a one-way hash function and fingerprint the PEM code. Then, each time you run it, you could check the fingerprint for modification. But Mallet could modify the fingerprint code at the same time he modifies the PEM code. You could fingerprint the fingerprint code, but Mallet could modify that as well. If Mallet can get access to your machine, he can subvert the security of PEM.

The moral is that you can never really trust a piece of software if you cannot trust the hardware it is running on. For most people, this kind of paranoia is unwarranted. For some, it is very real.

TIS-PEM

Trusted Information Systems, partially supported by the U.S. government Advanced Research Projects Agency, has designed and implemented a reference implementation of Privacy-Enhanced Mail (TIS/PEM). The implementation was developed for UNIX-based platforms but has also been ported to VMS. A DOS and MS Windows port is under development.

The PEM specifications currently indicate that RSA and DES are to be used in PEM. However, there is provision within the specifications for additional suites of algorithms to be used within the same framework, and whenever an algorithm is used, it is labeled with an identifier. In keeping with this architecture, the TIS/PEM system is modular and is written to accommodate multiple suites of algorithms.

Including support for other algorithms in TIS/PEM only requires a cryptographic library to perform the actual operation, a small set of routines to

interface TIS/PEM to the cryptographic library, and an entry in a table of cryptographic algorithms supported. In the Internet distribution of TIS/PEM, the RSA routines are provided via RSAREF.

Although the PEM specifications indicate a single certification hierarchy for use by the Internet, TIS/PEM supports the existence of multiple certification hierarchies. This is accomplished by allowing sites to specify a set of certificates that are to be considered valid, including all certificates issued by them. In addition, sites are not required to join the Internet hierarchy in order to acquire TIS/PEM.

TIS/PEM is currently available to all U.S. and Canadian organizations and citizens upon request (see contact information below). It will be distributed in source code form. Distribution via anonymous FTP is currently being investigated. Interested parties should contact:

Privacy-Enhanced Mail
Trusted Information Systems, Inc.
3060 Washington Road (Rte. 97)
Glenwood, MD 21738
Tel: (301) 854-6889
Fax: (301) 854-5363
Internet: pem-info@tis.com

RIPEM

RIPEM is a program, written by Mark Riorden, that implements the PEM protocols. Although technically not public domain, the program is publicly available and can be used royalty-free for personal, noncommercial applications. A license for its use is included with the documentation.

The code cannot be exported. To quote from the program's documentation:

> Please do not export the cryptographic code in this distribution outside the USA or Canada. This a personal request from me, the author of RIPEM, and a condition of your use of RIPEM.

Of course, U.S. government laws don't apply outside the United States, and people have ignored the export rules. RIPEM code has found its way outside the country and it is available worldwide.

RIPEM uses RSA Data Security's reference implementation of the RSA algorithm: RSA-REF. At this writing, RIPEM is not a complete implementation of the PEM protocols; it does not implement certificates for authenticating keys. Riorden plans to implement this feature, and it will be completed soon after this book is published.

Before writing RIPEM, Riorden wrote a similar program called RPEM. This was intended to be a public-domain electronic-mail encryption program. To try to avoid patent issues, Riorden used Rabin's algorithm (see Section 12.6). Public Key Partners claimed that their patents were broad enough to cover all of

public-key cryptography and threatened to sue; Riorden stopped distributing the program.

RPEM isn't really used anymore. It is not compatible with RIPEM. Since RIPEM can be used with the full blessing of Public Key Partners, there is no reason to use RPEM instead.

17.8 MESSAGE SECURITY PROTOCOL (MSP)

The Message Security Protocol (MSP) is the military equivalent of PEM. It was developed by the NSA in the late 1980s under the Secure Data Network System (SDNS) program. It is an X.400-compatible application-level protocol for securing electronic mail. It will be used for signing and encrypting messages in the Department of Defense's planned Defense Message System (DMS) network.

The Preliminary Message Security Protocol (PMSP), to be used for "unclassified but sensitive" messages, is a version of MSP adapted for use with both X.400 and TCP/IP. This protocol is also called "Mosaic."

Like PEM, MSP and PMSP software applications are flexibly designed to accommodate a variety of algorithms for a variety of security functions, including signing, hashing, and encryption. PMSP will work with the Capstone chip (see Section 17.11).

17.9 PRETTY GOOD PRIVACY (PGP)

Pretty Good Privacy (PGP) is a public-domain encryption program designed by Philip Zimmermann [908]. It uses IDEA for data encryption, RSA (with a 512-bit, 1024-bit, or 1280-bit key) for key management, and MD5 as a one-way hash function. PGP also compresses files before encryption.

PGP's random public keys uses a probabilistic primality tester, and gets its initial seeds from measuring the user's keyboard latency while typing. PGP generates random IDEA keys using the method delineated in ANSI X9.17, Appendix C [30], with IDEA as the symmetric algorithm instead of DES. PGP also encrypts the user's private key using a hashed pass phrase instead of a password.

PGP-encrypted messages have layered security. The only thing cryptanalysts can learn about an encrypted message is who the recipient is, assuming they know the recipient's key ID. Only after the recipient decrypts the message do they learn who signed the message, if it is signed. After verifying the signature they decrypt the message and learn what it says. Contrast this approach with PEM, which leaves quite a bit of information about the sender, recipient, and message in the unencrypted header.

The most interesting aspect of PGP is its distributed approach to key management (see Section 7.3). There are no key certification authorities. All users generate and distribute their own public key. Users can sign one another's public keys, adding extra confidence to the key's validity. Someone who signs another's public key becomes an introducer for that person. When users receive a new public

key, they examine the list of introducers who have signed the key. If one of the introducers is someone they trust, then they have reason to accept the new key as valid. If two of the introducers are people they trust marginally, then they have some reason to accept it as valid. This is all automatic: Tell PGP who you trust and how much, assess your own trust level, and proceed. PGP is well designed, nicely coded, and in the public domain. It's the closest you're likely to get to military-grade encryption.

17.10 CLIPPER

Clipper is an NSA-designed, tamper-resistant VLSI chip that uses the Skipjack encryption algorithm (see Section 11.12). Each chip has a special key, not needed for messages. It is used only to encrypt a copy of each user's message key. Anyone knowing the special key can decrypt wiretapped communications protected with this chip. This ensures government personnel the ability to conduct electronic surveillance. The claim is that only the government will know this key and will use it only if/when authorized by a court.

This special key is split (see Section 3.5) and stored in two key-escrow databases. According to the director of NIST [472]:

> A "key-escrow" system is envisioned that would ensure that the "Clipper Chip" is used to protect the privacy of law-abiding Americans. Each device containing the chip will have two unique "keys," numbers that will be needed by authorized government agencies to decode messages encoded by the device. When the device is manufactured, the two keys would be deposited separately in two "key-escrow" data bases established by the Attorney General. Access to these keys would be limited to government officials with legal authorization to conduct a wiretap.

At this writing, much about the Clipper chip, including the actual key-escrow mechanism, is undefined. The following section describes what I do know [472,653,876,270,56].

The Clipper chip is designed for the AT&T commercial secure voice products. The functionality of the chip is specified by NSA, the logic is designed by Mykotronx, Inc., and the chip is fabricated by VLSI, Inc. (at this writing there is talk of a second source for the chips).

Each chip is uniquely programmed before being sold to customers. The chip programming equipment writes the following information into a special memory (called VROM or VIA-Link) on the chip:

- A serial number, unique to the chip
- A unit key, unique to the chip
- A family key, common to a family of chips
- Any specialized control software

Upon generation (or entry) of a session key in the chip, the chip performs the following actions:

(1) Encrypts the 80-bit session key under the unit key, producing an 80-bit intermediate result;

(2) Concatenates the 80-bit result with the 25-bit serial number and a 23-bit authentication pattern (total of 128 bits);

(3) Encrypts this 128 bits with the family key to produce a 128-bit cipher block chain called the Law Enforcement Field (LEF);

(4) Transmits the LEF at least once to the intended receiving Clipper chip;

(5) The two communicating Clipper chips use this field together with a random IV to establish synchronization.

(6) Once synchronized, the Clipper chips use the session key to encrypt/decrypt data in both directions.

The point of this complicated process is to allow someone with the unique serial number and unique unit key to decrypt messages encrypted with this chip. This is the back door that the government is embedding into the system; it is not a back door to the Skipjack algorithm, but a back door to Clipper's key-management scheme. To prevent someone from circumventing this scheme, the chips can be programmed not to enter secure mode if the LEF field has been tampered with (e.g., modified, superencrypted, replaced). The chip also is resistant to reverse-engineering "against a very sophisticated, well funded adversary" [653].

There are enormous privacy issues associated with this scheme. Both the Computer Professionals for Social Responsibility and the Electronic Frontier Foundation are actively campaigning against any key-escrow mechanism that gives the government the right to eavesdrop on citizens.

17.11 CAPSTONE

CAPSTONE is an NSA-developed VLSI cryptographic chip that implements the Skipjack algorithm as well as the same key-escrow features as the Clipper chip. In addition, the Capstone chip includes the following functions [654]:

- The Digital Signature Algorithm (DSA).
- The Secure Hashing Algorithm (SHA).
- A general purpose exponentiation algorithm.
- A general purpose, random number generator that uses a pure noise source.

A key exchange algorithm (probably Diffie-Hellman) is also programmable on the chip and uses the above four functions.