CHAPTER 3

Basic Protocols

3.1 KEY EXCHANGE

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

Key Exchange with Symmetric Cryptography

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

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

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

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

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

Key Exchange with Public-Key Cryptography

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

(1) Alice gets Bob's public key from the KDC.

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

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

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

Man-in-the-Middle Attack

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

(1) Alice sends Bob her public key. Mallory intercepts this key and sends Bob his own public key.

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

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

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

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

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

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

Interlock Protocol

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

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

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

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

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

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

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

Key Exchange with Digital Signatures

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

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

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

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

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

Key and Message Transmission

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

(1) Alice generates a random session key, K, and encrypts M using K.
$E_K\{M\}$
(2) Alice gets Bob's public key from the database.
(3) Alice encrypts K with Bob's public key.
$E_B\{K\}$
(4) Alice sends both the encrypted message and encrypted session key to Bob.
$E_K\{M\}, E_B\{K\}$
For added security against man-in-the-middle attacks, Alice can sign the transmission.
(5) Bob decrypts Alice's session key, K, using his private key.
(6) Bob decrypts Alice's message using the session key.

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

Key and Message Broadcast

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

(1) Alice generates a random session key, K, and encrypts M using K.
$E_K\{M\}$
(2) Alice gets Bob's, Carol's, and Dave's public keys from the database.
(3) Alice encrypts K with Bob's public key, encrypts K with Carol's public key, and then encrypts K with Dave's public key.
$E_B\{K\}, E_C\{K\}, E_D\{K\}$
(4) Alice broadcasts the encrypted message and all the encrypted keys to anybody who cares to receive it.
$E_B\{K\}, E_C\{K\}, E_D\{K\}, E_K\{M\}$
(5) Only Bob, Carol, and Dave can decrypt the key, K, each using his or her private key.
(6) Only Bob, Carol, and Dave can decrypt Alice's message using K.

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

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

3.2 AUTHENTICATION

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

Authentication Using One-Way Functions

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

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

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

Dictionary Attacks and Salt

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

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

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

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

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

SKEY

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

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

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

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

Authentication Using Public-Key Cryptography

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

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

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

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

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

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

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

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

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

Mutual Authentication Using the Interlock Protocol

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

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

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

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

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

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

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

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

SKID

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

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

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

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

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

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

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

(4) Alice sends Bob:

$$H_K\{R_B,A\}$$

A is Alice's name.

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

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

Message Authentication

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

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

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

3.3 AUTHENTICATION AND KEY EXCHANGE

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

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

Wide-Mouth Frog

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

TABLE 3.1
Symbols used in authentication and key exchange protocols

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

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

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

$$A, E_A\{T_A, B, K\}$$

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

$$E_B\{T_B, A, K\}$$

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

Yahalom

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

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

$$A, R_A$$

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

$$B, E_B\{A, R_A, R_B\}$$

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

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

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

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

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

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

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

Needham-Schroeder

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

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

$$A,B,R_A$$

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

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

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

$$E_B(K,A)$$

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

$$E_K(R_B)$$

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

$$E_K(R_B - 1)$$

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

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

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

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

(1) Mallory sends Bob the following message:
$E_B(K,A)$

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

(3) Mallory intercepts the message, decrypts it with K, and sends Bob:
$E_K(R_B - 1)$

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

Now, Mallory has Bob convinced that he is Alice.

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

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

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

Otway-Rees

This protocol also uses symmetric cryptography [1224].

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

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

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

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

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

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

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

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

Kerberos

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

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

$$A, B$$

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

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

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

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

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

$$E_K(T + 1)$$

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

Neuman-Stubblebine

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

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

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

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

$$A, R_A$$

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

$$B, R_B, E_B\{A, R_A, T_B\}$$

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

$$E_A\{B, R_A, K, T_B\}, E_A\{A, K, T_B\}, R_B$$

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

$$E_B\{A, K, T_B\}, E_K\{R_B\}$$

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

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

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

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

$$E_B\{A, K, T_B\}, R'_A$$

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

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

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

$$E_K(R'_B)$$

The new random numbers prevent replay attacks.

DASS

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

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

$$B$$

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

$$S_T(B, K_B)$$

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

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

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

$$A$$

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

$$S_T(A, K_A)$$

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

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

$$E_K(T_B)$$

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

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

Denning-Sacco

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

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

$$A,B$$

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

$$S_T\{B,K_B\},S_T\{A,K_A\}$$

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

$$E_B\{S_A\{K,T_A\}\},S_T\{B,K_B\},S_T\{A,K_A\}$$

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

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

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

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

$$B,C$$

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

$$S_T\{B,K_B\},S_T\{C,K_C\}$$

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

$$E_C\{S_A\{K,T_A\}\},S_T\{A,K_A\},S_T\{C,K_C\}$$

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

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

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

$$E_B\{S_A\{A,B,K,T_A\}\},S_T\{A,K_A\},S_T\{B,K_B\}$$

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

Woo-Lam

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

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

A,B

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

$S_T\{K_B\}$

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

$E_{K_B}\{A,R_A\}$

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

$A,B,E_{K_T}\{R_A\}$

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

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

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

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

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

$E_K\{R_B\}$

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

Other Protocols

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

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

Lessons Learned

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

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

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

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

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

3.4 Formal Analysis of Authentication and Key-Exchange Protocols

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

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

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

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

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

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

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

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

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

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

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

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

Another rule is the nonce-verification rule:

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

There are four steps in BAN analysis:

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

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

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

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

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

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

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

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

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

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

— Learn (Eve, M). (Eve learns M.)

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

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

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

— If Bob accepted message M from Alice at some point in the past, then Eve did not learn M at some point in the past.
— If Bob accepted message M from Alice in Bob's local round N, then Alice sent M to Bob as a response to a query in Bob's local round N.

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

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

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

3.5 MULTIPLE-KEY PUBLIC-KEY CRYPTOGRAPHY

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

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

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

TABLE 3.2
Three-Key Key Distribution

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

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

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

Broadcasting a Message

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

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

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

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

TABLE 3.3
Three-Key Message Encryption

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

3.6 SECRET SPLITTING

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

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

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

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

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

$$M \oplus R = S$$

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

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

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

$$R \oplus S = M$$

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

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

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

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

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

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

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

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

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

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

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

3.7 Secret Sharing

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

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

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

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

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

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

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

Secret Sharing with Cheaters

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

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

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

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

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

Secret Sharing without Trent

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

Sharing a Secret without Revealing the Shares

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

Verifiable Secret Sharing

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

Secret-Sharing Schemes with Prevention

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

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

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

Secret Sharing with Disenrollment

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

3.8 Cryptographic Protection of Databases

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

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

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

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

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

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

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

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